We now repeat the process for β.

$$\begin{aligned}\frac{\partial}{\partial\beta}\log P(D|\alpha,\beta,M) &= \frac{\partial}{\partial\beta}F(\mathbf{w}^{MP}) - \frac{\partial}{\partial\beta}\frac{1}{2}\log\det(\mathbf{H}^{MP}) + \frac{\partial}{\partial\beta}\frac{N}{2}\log\beta \\ &= \frac{\partial}{\partial\beta}(\beta E_D(\mathbf{w}^{MP})) - \frac{\gamma}{2\beta^{MP}} + \frac{N}{2\beta^{MP}} \\ &= -E_D(\mathbf{w}^{MP}) - \frac{\gamma}{2\beta^{MP}} + \frac{N}{2\beta^{MP}} = 0\end{aligned}$$

Rearranging terms,

$$\begin{aligned}E_D(\mathbf{w}^{MP}) &= \frac{N}{2\beta^{MP}} - \frac{\gamma}{2\beta^{MP}} \\ \beta^{MP} &= \frac{N-\gamma}{2E_D(\mathbf{w}^{MP})}\end{aligned}$$

P13.4 Demonstrate that extrapolation can occur in a region that is surrounded by training data.

Consider the function displayed in Figure 13.3. In that example, extrapolation occurred in the upper left region of the input space, because all of the training data was in the lower right. Let's provide training data around the outside of the input space, but without data in the region

$$-1.5 < p_1 < 1.5 \qquad -1.5 < p_2 < 1.5 .$$

The training data is distributed as shown in Figure P13.4.

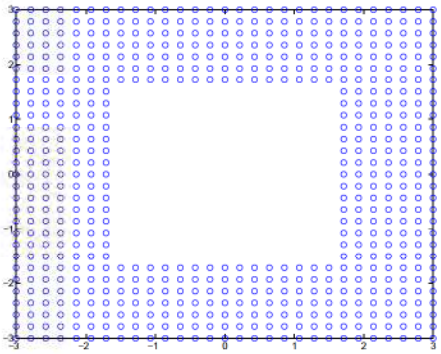


Figure P13.4 Training Data Locations

The result of the training is shown in Figure P13.5. The neural network approximation significantly overestimates the true function in the region without training data, even though surrounded by regions with training data. In addition, this result is random. With a different set of initial random weights, the network might underestimate the true function in this region. Extrapolation occurs because there is a significantly large region without training data. When the input space is of high dimension, it can be very difficult to tell when a network is extrapolating. It cannot be done by simply checking the individual ranges of each input variable.

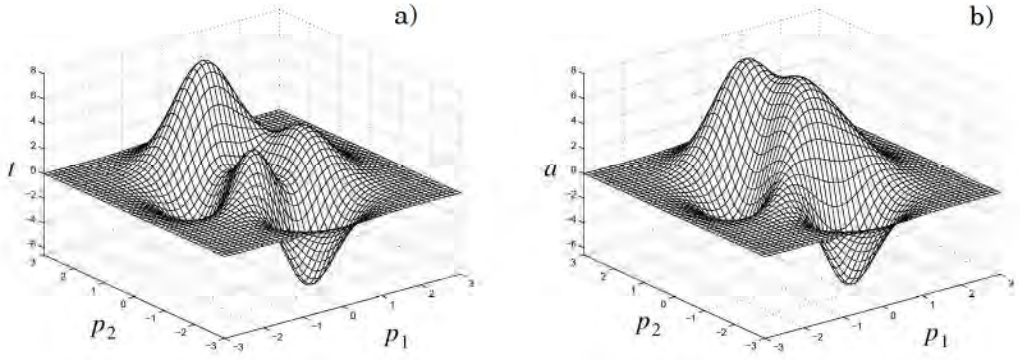


Figure P13.5 Function (a) and Neural Network Approximation (b)

P13.5 Consider the example starting on page 13-23. Find the effective number of parameters if $\rho = 1$.

To find the effective number of parameters, we can use Eq. (13.55):

$$\gamma = \sum_{i=1}^{n} \frac{\beta\lambda_i}{\beta\lambda_i + 2\alpha}.$$

We earlier found the eigenvalues to be $\lambda_1 = 1$, $\lambda_2 = 3$. The regularization parameter is

$$\rho = \frac{\alpha}{\beta} = 1.$$

We can rewrite γ in terms of ρ as follows

$$\gamma = \sum_{i=1}^{n} \frac{\beta\lambda_i}{\beta\lambda_i + 2\alpha} = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda_i + 2\frac{\alpha}{\beta}} = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda_i + 2\rho}.$$

Substituting our numbers, we find

$$\gamma = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda_i + 2\rho} = \frac{1}{1+2} + \frac{3}{3+2} = \frac{1}{3} + \frac{3}{5} = \frac{14}{15}.$$

Therefore, we are using approximately one of the two available parameters. The network has two parameters: $w_{1,1}$ and $w_{1,2}$. The parameter we are using is not one of these two, but rather a combination. As we can see from Figure 13.11, we move in the direction of the second eigenvector:

$$\mathbf{v}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

which means that we are changing $w_{1,1}$ and $w_{1,2}$ by the same amount. Although there are two parameters, we are effectively using only one. Since $\mathbf{v}_2$ is the eigenvector with the largest eigenvalue, we move in that direction to obtain the greatest reduction in the squared error.

P13.6 Demonstrate overfitting with polynomials. Consider fitting a polynomial

$$g_k(p) = x_0 + x_1 p + x_2 p^2 + \ldots + x_k p^k$$

to a set of data $\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\}$ **so as to minimize the following squared error performance function.**

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (t_q - g_k(p_q))^2$$

First, we want to express the problem in matrix form. Define the following vectors.

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_Q \end{bmatrix} \quad \mathbf{G} = \begin{bmatrix} 1 & p_1 & \ldots & p_1^k \\ 1 & p_2 & \ldots & p_2^k \\ \vdots & \vdots & & \vdots \\ 1 & p_Q & \ldots & p_Q^k \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_k \end{bmatrix}$$

We can then write the performance index as follows.

$$F(\mathbf{x}) = [\mathbf{t} - \mathbf{G}\mathbf{x}]^T [\mathbf{t} - \mathbf{G}\mathbf{x}] = \mathbf{t}^T\mathbf{t} - 2\mathbf{x}^T\mathbf{G}^T t + \mathbf{x}^T\mathbf{G}^T\mathbf{G}\mathbf{x}$$

To locate the minimum, we take the gradient and set it equal to zero.

$$\nabla F(\mathbf{x}) = -2\mathbf{G}^T t + 2\mathbf{G}^T\mathbf{G}\mathbf{x} = 0$$

Solving for the weights, we obtain the least squares solution (maximum likelihood for the Gaussian noise case).

$$[\mathbf{G}^T\mathbf{G}]\mathbf{x}^{ML} = \mathbf{G}^T t \qquad \Rightarrow \qquad \mathbf{x}^{ML} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T t$$

To demonstrate the operation of the polynomial fitting, we will use the simple linear function $t = p$. To create the data set, we will sample the function at five different points and will add noise as follows

$$t_i = p_i + \varepsilon_i, \; p = \{-1, -0.5, 0, 0.5, 1\},$$

where ε_i has a uniform density with range $[-0.25, 0.25]$. The code below shows how to generate the data and fit a 4th order polynomial. The results of fitting 2nd and 4th order polynomials are shown in Figure P13.6. The 4th order polynomial has five parameters, which allow it to exactly fit the five noisy data points, but it doesn't produce an accurate approximation of the true function.

```
» 2 + 2
ans =
     4
```

```
p = -1:.5:1;
t = p + 0.5*(rand(size(p))-0.5);
Q = length(p);
ord = 4;
G = ones(Q,1);
for i=1:ord,
    G = [G (p').^i];
end
x = (G'*G)\G'*t'; % Could also use x = G\t';
```

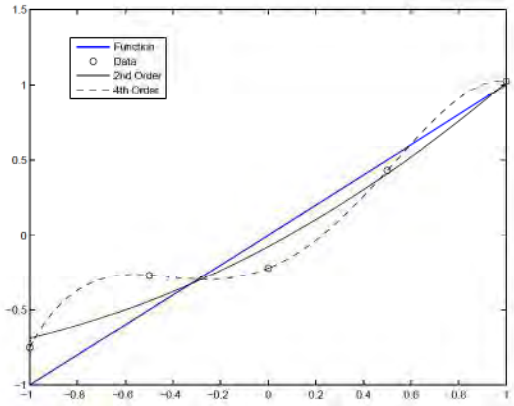


Figure P13.6 Polynomial Approximations to a Straight Line

Epilogue

The focus of this chapter has been the development of algorithms for training multilayer neural networks so that they generalize well. A network that generalizes well will perform as well in new situations as it performs on the data for which it was trained.

The basic approach to producing networks that generalize well is to find the simplest network that can represent the data. A simple network is one that has a small number of weights and biases.

The two methods that we presented in this chapter, early stopping and regularization, produce simple networks by constraining the weights, rather than by reducing the number of weights. We showed in this chapter that constraining the weights is equivalent to reducing the number of weights.

Chapter 23 presents a case study that uses Bayesian regularization to prevent overfitting in a practical function approximation problem. Chapter 25 presents a case study that uses early stopping to prevent overfitting in a practical pattern recognition problem.

Further Reading

[AmMu97] S. Amari, N. Murata, K.-R. Muller, M. Finke, and H. H. Yang, "Asymptotic Statistical Theory of Overtraining and Cross-Validation," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, 1997.

When using early stopping, it is important to decide on the number of data points to place in the validation set. This paper provides a theoretical basis for the choice of validation set size.

[FoHa97] D. Foresee and M. Hagan, "Gauss-Newton Approximation to Bayesian Learning," *Proceedings of the 1997 International Joint Conference on Neural Networks*, vol. 3, pp. 1930 - 1935, 1997.

This paper describes a method for implementing Bayesian regularization by using the Gauss-Newton approximation to the Hessian matrix.

[GoLa98] C. Goutte and J. Larsen, "Adaptive Regularization of Neural Networks Using Conjugate Gradient," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 2, pp. 1201-1204, 1998.

When using regularization, the important step is setting the regularization parameter. This paper describes a procedure for setting the regularization parameter to minimize the validation set error.

[MacK92] D. J. C. MacKay, "Bayesian Interpolation," *Neural Computation*, vol. 4, pp. 415-447, 1992.

Bayesian approaches have been used for many years in statistics. This paper presents one of the first developments of a Bayesian framework for training neural networks. MacKay followed this paper with many others describing refinements of the approach.

[Sarle95] W. S. Sarle, "Stopped training and other remedies for overfitting," In *Proceedings of the 27th Symposium on Interface*, 1995.

This is one of the early papers on the use of early stopping with a validation set to prevent overfitting. The paper describes simulation results comparing early stopping with other methods for improving generalization.

[SjLj94] J. Sjoberg and L. Ljung, "Overtraining, regularization and searching for minimum with application to neural networks," Linkoping University, Sweden, Tech. Rep. LiTH-ISY-R-1567, 1994.

This report explains how early stopping and regularization are approximately equivalent processes. It demonstrates that the number of iterations of training is inversely proportional to the regularization parameter.

[Tikh63] A. N. Tikhonov, "The solution of ill-posed problems and the regularization method," *Dokl. Acad. Nauk USSR*, vol. 151, no. 3, pp. 501-504, 1963.

Regularization is a method by which a squared error performance index is augmented by a penalty on the complexity of the approximating function. This is the original paper that introduced the concept of regularization. The penalty involved the derivatives of the approximating function.

[WaVe94] C. Wang, S. S. Venkatesh, and J. S. Judd, "Optimal Stopping and Effective Machine Complexity in Learning," *Advances in Neural Information Processing Systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6, pp. 303-310, 1994.

This paper describes how the effective number of network parameters changes during the training process and how the generalization capability of the network can be improved by stopping the training early.

Exercises

E13.1 Consider fitting a polynomial (kth order)

$$g_k(p) = x_0 + x_1 p + x_2 p^2 + \ldots + x_k p^k$$

to a set of data $\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\}$. It has been proposed that minimizing a performance index that penalizes the derivatives of the polynomial will provide improved generalization. Investigate the relationship between this technique and regularization using squared weights.

i. Derive the least squares solution for the weights x_i, which minimizes the following squared error performance index. (See Solved Problem P13.6.)

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (t_q - g_k(p_q))^2$$

ii. Derive the regularized least squares solution, with a squared weight penalty.

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (t_q - g_k(p_q))^2 + \rho \sum_{i=0}^{k} x_i^2$$

iii. Derive a solution for the weights that minimizes a sum of the squared error plus a sum of squared derivatives.

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (t_q - g_k(p_q))^2 + \rho \sum_{i=1}^{Q} \left[\frac{d}{dp} g_k(p_q)\right]^2$$

iv. Derive a solution for the weights that minimizes a sum of the squared error plus a sum of squared second derivatives.

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (t_q - g_k(p_q))^2 + \rho \sum_{i=1}^{Q} \left[\frac{d^2}{dp^2} g_k(p_q)\right]^2$$



E13.2 Write a MATLAB program to implement the solutions you found in E13.1 i. through iv. Using the following data points, adjust the ρ values to obtain the best results. Use $k = 8$ for all cases. Plot the data points, the noise-free function ($t = p$) and the polynomial approximation in each case. Compare the four approximations. Which do you think produces the best results? Which cases produce similar results?

$$t_i = p_i + \varepsilon_i,\ p = \{-1, -0.5, 0, 0.5, 1\},$$

where ε_i has a uniform density with range $[-0.1, 0.1]$ (use the rand command in MATLAB).

E13.3 Consider fitting a polynomial (1st order), $g_1(p) = x_0 + x_1 p$, to the following data set:

$$\left\{p_1 = \left[1\right], t_1 = \left[4\right]\right\}, \left\{p_2 = \left[2\right], t_2 = \left[6\right]\right\}.$$

i. Find the least squares solutions for the weights x_0 and x_1 that minimize the following sum squared error performance index:

$$F(\mathbf{x}) = \sum_{q=1}^{2} (t_q - g_1(p_q))^2.$$

ii. Find the regularized least squares solution for the weights x_0 and x_1 when the following squared weight penalty is used:

$$F(\mathbf{x}) = \sum_{q=1}^{2} (t_q - g_1(p_q))^2 + \sum_{i=0}^{1} x_i^2.$$

E13.4 Investigate the extrapolation characteristics of neural networks and polynomials. Consider the problem described in E11.25, where a sine wave is fit over the range $-2 \le p \le 2$. Select 11 training points evenly spaced over this interval.

```
» 2+2
ans =
     4
```

i. After fitting the 1-2-1 neural network over this range, plot the actual sine function and the neural network approximation over the range $-4 \le p \le 4$.

ii. Fit a fifth-order polynomial (which has the same number of free parameters as the 1-2-1 network) to the sine wave over the range $-2 \le p \le 2$ (using your results from E13.1 i.). Plot the actual function and the polynomial approximation over the range $-4 \le p \le 4$.

iii. Discuss the extrapolation characteristics of the neural network and the polynomial.

E13.5 Suppose that we have a random variable t that is distributed according to the following density function. We take a series of Q independent samples of the random variable. Find the maximum likelihood estimate of x (x^{ML}).

$$f(t|x) = \frac{t}{x^2}\exp\left(-\frac{t}{x}\right) \qquad t \geq 0$$

E13.6 For the random variable t given in E13.5, suppose that x is a random variable with the following prior density function. Find the most probable estimate of x (x^{MP}).

$$f(x) = \exp(-x) \qquad x \geq 0$$

E13.7 Repeat E13.6 for the following prior density function. Under what conditions will $x^{MP} = x^{ML}$?

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_x}\exp\left(-\frac{(x-\mu_x)^2}{2\sigma_x^2}\right)$$

E13.8 In the signal plus noise example given in Solved Problem P13.2, find x^{MP} for the following prior density functions.

i. $f(x) = \exp(-x) \qquad x \geq 0$

ii. $f(x) = \frac{1}{2}\exp(-|x-\mu|)$

E13.9 Suppose that we have a random variable t that is distributed according to the following density function. We take a series of $Q = 2$ independent samples of the random variable.

$$f(t|x) = \begin{cases} \exp(-(t-x)) & t \geq x \\ 0 & t < x \end{cases}$$

i. Find the likelihood function $f(t_1, t_2|x)$, and sketch versus x.

ii. Suppose that the two measurements are $t_1 = 1$ and $t_2 = 1$. Find the maximum likelihood estimate of x (x^{ML}).

For the random variable t above, suppose that x is a random variable with the following prior density function.

$$f(x) = \frac{1}{\sqrt{\pi/2}}\exp(-2x^2)$$

iii. Sketch the posterior density $f(x|t_1, t_2)$. (You do not need to compute the denominator, just find the general shape. Assume the same measurements from part ii.)

iv. Find the most probable estimate of x (x^{MP}).

E13.10 We have a coin that is not fair. (The probability of heads is not the same as the probability of tails.) We want to estimate the probability of heads (x).

i. If we flip the coin 10 times, the probability of getting exactly t heads, given that the probability of heads is x, is given below. Find the maximum likelihood estimate of x (x^{ML}). (Hint: Take the natural log of $p(t|x)$ before finding the maximum.) Is it reasonable? Explain.

$$p(t|x) = \binom{10}{t} x^t (1-x)^{(10-t)}, \text{ where } \binom{10}{t} = \frac{10!}{t!(10-t)}$$

ii. Assume that the probability of heads, x, is a random variable with the following prior density function. Find the most probable estimate of x (x^{MP}). (Hint: Take the natural log of $p(t|x)p(x)$ before finding the maximum.) Explain why x^{ML} is different than x^{MP}.

$$p(x) = 12x^2(1-x), \; 0 \le x \le 1.$$

E13.11 Suppose that the prior density in the level I Bayesian analysis (see page 13-12) has nonzero mean, μ_x. Find the new performance index.

E13.12 Suppose that we have the following inputs and targets:

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, t_1 = 1\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix} -2 \\ -1 \end{bmatrix}, t_2 = 3\right\}$$

We want to train a single-layer linear network without a bias on this training set. Assume that each input vector occurs with equal probability. We will train the network using the regularized performance index of Eq. (13.34).

i. Find $\mathbf{x}^{MP}$ and the effective number of parameters γ, if $\rho = 1$.

ii. Find $\mathbf{x}^{MP}$ and γ as $\rho \to \infty$. Explain the difference between $\mathbf{x}^{MP}$ and $\mathbf{x}^{ML}$. (Your answer should be specific to this problem—not a general discussion of the difference between $\mathbf{x}^{MP}$ and $\mathbf{x}^{ML}$.)

E13.13 Suppose we have the following input pattern and target:

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, t_1 = 1\right\}.$$

This pattern is used to train a single layer, linear network with no bias.

i. The network is to be trained with a regularized performance index, with the regularization parameter set to $\rho = \alpha/\beta = 1/2$. Find an expression for the regularized performance index.

ii. Find x^{MP} and the effective number of parameters γ.

iii. Sketch the contour plot of the regularized performance index.

iv. Find the maximum stable learning rate if steepest descent is used to train the network.

v. Find the initial direction for the steepest descent algorithm trajectory, if both initial weights are set to zero.

vi. Sketch an approximate complete trajectory (on your contour plot from part iii.) for the steepest algorithm, with very small learning rate, from the initial conditions where both weights are set to zero. Explain your procedure for sketching the trajectory.

E13.14 Suppose that we have a single layer, linear network with no bias. The input/target pairs of the training set are given by

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, t_1 = -2\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}, t_2 = 2\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}2\\1\end{bmatrix}, t_3 = 4\right\},$$

where each pair occurs with equal probability. We want to minimize the regularized performance index of Eq. (13.34).

i. Find the effective number of parameters γ, for $\rho = 1$.

ii. Starting with zero initial weights, approximately how many iterations of the steepest descent algorithm would need to be made on the mean square performance index E_D to produce results that would be equivalent to minimizing the regularized performance index with $\rho = 1$? Assume a learning rate of $\alpha = 0.01$.

E13.15 Repeat E11.25, but modify your program to use early stopping and to use 30 neurons. Select 10 training points and 5 validation points. Add noise to the validation and testing points that is uniformly distributed between -0.1 and 0.1 (using the MATLAB function `rand`). Measure the mean square error of the trained network on a testing set consisting of 20 equally-spaced points of the noise-free function. Try 10 different random sets of training and validation data. Compare the results with early-stopping with the results without early stopping.



E13.16 Repeat E13.15, but use regularization instead of early stopping. This will require modifying your program to compute the gradient of the regularized performance index. Add the standard gradient of the squared error, which is computed by the standard backpropagation algorithm, to the gradient of ρ times the squared weights. Try three different values of ρ. Compare these results with the early stopping results.



E13.17 Consider again the problem described in E10.4

i. Find the regularized performance index for $\rho = 0, 1, \infty$. Sketch the contour plot in each case. Indicate the location of the optimal weights in each case.

ii. Find the effective number of parameters for $\rho = 0, 1, \infty$.

iii. Starting with zero initial weights, approximately how many iterations of the steepest descent algorithm would need to be made on the mean square performance index to produce results that would be equivalent to minimizing the regularized performance index with $\rho = 1$? Assume a learning rate of $\alpha = 0.01$.



iv. Write a MATLAB M-file to implement the steepest descent algorithm to minimize the mean square error performance index that you found in part i. (This is a quadratic function.) Start the algorithm with zero initial conditions, and use a learning rate of $\alpha = 0.01$. Sketch the trajectory on a contour plot of the mean square error (the contour plot was found in E10.4). Verify that at the iteration you computed in part iii., the weights are close to the same values you found to minimize the regularized performance index with $\rho = 1$ in part i.

14 Dynamic Networks

Objectives	14-1
Theory and Examples	14-2
Layered Digital Dynamic Networks	14-3
Example Dynamic Networks	14-5
Principles of Dynamic Learning	14-8
Dynamic Backpropagation	14-12
Preliminary Definitions	14-12
Real Time Recurrent Learning	14-12
Backpropagation-Through-Time	14-22
Summary and Comments on Dynamic Training	14-30
Summary of Results	14-34
Solved Problems	14-36
Epilogue	14-45
Further Reading	14-46
Exercises	14-47

Objectives

Neural networks can be classified into static and dynamic categories. The multilayer network that we have discussed in the last three chapters is a static network. This means that the output can be calculated directly from the input through feedforward connections. In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs or states of the network. For example, the adaptive filter networks we discussed in Chapter 10 are dynamic networks, since the output is computed from a tapped delay line of previous inputs. The Hopfield network we discussed in Chapter 3 is also a dynamic network. It has recurrent (feedback) connections, which means that the current output is a function of outputs at previous times.

We will begin this chapter with a brief introduction to the operation of dynamic networks, and then we will describe how these types of networks can be trained. The training will be based on optimization algorithms that use gradients (as in steepest descent and conjugate gradient algorithms) or Jacobians (as in Gauss-Newton and Levenberg-Marquardt algorithms) These algorithms were described in Chapters 10, 11 and 12 for static networks. The difference between the training of static and dynamic networks is in the manner in which the gradient or Jacobian is computed. In this chapter, we will present methods for computing gradients for dynamic networks.

Theory and Examples

Dynamic Networks

Dynamic networks are networks that contain delays (or integrators, for continuous-time networks) and that operate on a sequence of inputs. (In other words, the ordering of the inputs is important to the operation of the network.) These dynamic networks can have purely feedforward connections, like the adaptive filters of Chapter 10, or they can also have some feedback (*recurrent*) connections, like the Hopfield network of Chapter 3. Dynamic networks have memory. Their response at any given time will depend not only on the current input, but on the history of the input sequence.

Recurrent

Because dynamic networks have memory, they can be trained to learn sequential or time-varying patterns. Instead of approximating functions, like the static multilayer perceptron network of Chapter 11, a dynmic network can approximate a dynamic system. This has applications in such diverse areas as control of dynamic systems, prediction in financial markets, channel equalization in communication systems, phase detection in power systems, sorting, fault detection, speech recognition, learning of grammars in natural languages, and even the prediction of protein structure in genetics.

Dynamic networks can be trained using the standard optimization methods that we have discussed in Chapters 9 through 12. However, the gradients and Jacobians that are required for these methods cannot be computed using the standard backpropagation algorithm. In this chapter we will present the dynamic backpropagation algorithms that are required for computing the gradients for dynamic networks.

There are two general approaches (with many variations) to gradient and Jacobian calculations in dynamic networks: backpropagation-through-time (BPTT) [Werb90] and real-time recurrent learning (RTRL) [WiZi89]. In the BPTT algorithm, the network response is computed for all time points, and then the gradient is computed by starting at the last time point and working backward in time. This algorithm is efficient for the gradient calculation, but it is difficult to implement on-line, because the algorithm works backward in time from the last time step.

In the RTRL algorithm, the gradient can be computed at the same time as the network response, since it is computed by starting at the first time point, and then working forward through time. RTRL requires more calculations than BPTT for calculating the gradient, but RTRL allows a convenient framework for on-line implementation. For Jacobian calculations, the RTRL algorithm is generally more efficient than the BPTT algorithm.

In order to more easily present general BPTT and RTRL algorithms, it will be helpful to introduce modified notation for networks that can have recurrent connections. In the next section we will introduce this notation, and then the remainder of the chapter will present general BPTT and RTRL algorithms for dynamic networks.

Layered Digital Dynamic Networks

LDDN

In this section we want to introduce the neural network framework that we will use to represent general dynamic networks. We call this framework Layered Digital Dynamic Networks (LDDN). It is an extension of the notation that we have used to represent static multilayer networks. With this new notation, we can conveniently represent networks with multiple recurrent (feedback) connections and tapped delay lines.

To help us introduce the LDDN notation, consider the example dynamic network given in Figure 14.1.

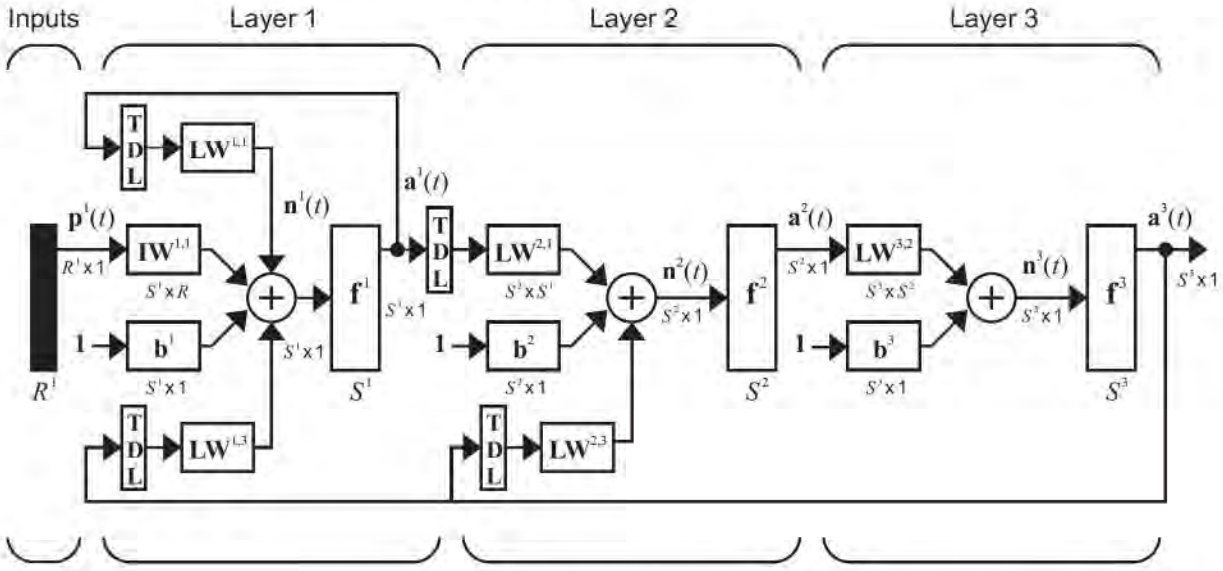


Figure 14.1 Example Dynamic Network

The general equations for the computation of the net input $\mathbf{n}^m(t)$ for layer m of an LDDN are

$$\mathbf{n}^m(t) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d) + \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \mathbf{b}^m \tag{14.1}$$

Input Weight
Layer Weight

where $\mathbf{p}^l(t)$ is the l^{th} input vector to the network at time t, $\mathbf{IW}^{m,l}$ is the *input weight* between input l and layer m, $\mathbf{LW}^{m,l}$ is the *layer weight* between layer l and layer m, $\mathbf{b}^m$ is the bias vector for layer m, $DL_{m,l}$ is the set of all delays in the tapped delay line between Layer l and Layer m, $DI_{m,l}$ is the set of all delays in the tapped delay line between Input l and Layer m,

I_m is the set of indices of input vectors that connect to layer m, and L_m^f is the set of indices of layers that directly connect *forward* to layer m. The output of layer m is then computed as

$$\mathbf{a}^m(t) = \mathbf{f}^m(\mathbf{n}^m(t)). \tag{14.2}$$

Compare this with the static multilayer network of Eq. (11.6). LDDN networks can have several layers connecting to layer m. Some of the connections can be recurrent through tapped delay lines. An LDDN can also have multiple input vectors, and the input vectors can be connected to any layer in the network; for static multilayer networks, we assumed that the single input vector connected only to Layer 1.

With static multilayer networks, the layers were connected to each other in numerical order. In other words, Layer 1 was connected to Layer 2, which was connected to Layer 3, etc. Within the LDDN framework, any layer can connect to any other layer, even to itself. However, in order to use Eq. (14.1), we need to compute the layer outputs in a specific order. The order in which the layer outputs must be computed to obtain the correct network output is called the *simulation order*. (This order need not be unique; there may be several valid simulation orders.) In order to backpropagate the derivatives for the gradient calculations, we must proceed in the opposite order, which is called the *backpropagation order*. In Figure 14.1, the standard numerical order, 1-2-3, is the simulation order, and the backpropagation order is 3-2-1.

Simulation Order

Backpropagation Order

As with the multilayer network, the fundamental unit of the LDDN is the layer. Each layer in the LDDN is made up of five components:

1. a set of weight matrices that come into that layer (which may connect from other layers or from external inputs),
2. any tapped delay lines (represented by $DL_{m,l}$ or $DI_{m,l}$) that appear at the input of a set of weight matrices (Any set of weight matrices can be preceded by a TDL. For example, Layer 1 of Figure 14.1 contains the weights $\mathbf{LW}^{1,3}(d)$ and the corresponding TDL.),
3. a bias vector,
4. a summing junction, and
5. a transfer function.

The output of the LDDN is a function not only of the weights, biases, and current network inputs, but also of some layer outputs at previous points in time. For this reason, it is not a simple matter to calculate the gradient of the network output with respect to the weights and biases. The weights and biases have two different effects on the network output. The first is the direct effect, which can be calculated using the standard backpropagation algorithm from Chapter 11. The second is an indirect effect, since some of

the inputs to the network are previous outputs, which are also functions of the weights and biases. The main development of the next two sections is a general gradient calculation for arbitrary LDDNs.

Example Dynamic Networks



Before we introduce dynamic training, let's get a feeling for the types of responses we can expect to see from dynamic networks. Consider first the feedforward dynamic network shown in Figure 14.2.

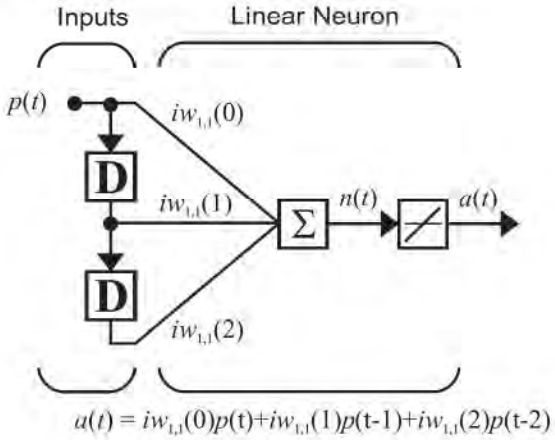


Figure 14.2 Example Feedforward Dynamic Network

This is an ADALINE filter, which we discussed in Chapter 10 (see Figure 10.5). Here we are representing it in the LDDN framework. The network has a TDL on the input, with $DI_{1,1} = \{0, 1, 2\}$. To demonstrate the operation of this network, we will apply a square wave as input, and we will set all of the weight values equal to 1/3:

$$iw_{1,1}(0) = \frac{1}{3},\ iw_{1,1}(1) = \frac{1}{3},\ iw_{1,1}(2) = \frac{1}{3}. \tag{14.3}$$

The network response is calculated from:

$$\begin{aligned} \mathbf{a}(t) = \mathbf{n}(t) &= \sum_{d=0}^{2} \mathbf{IW}(d)\mathbf{p}(t-d) \\ &= n_1(t) = iw_{1,1}(0)p(t) + iw_{1,1}(1)p(t-1) + iw_{1,1}(2)p(t-2) \end{aligned} \tag{14.4}$$

where we have left off the superscripts on the weight and the input, since there is only one input and only one layer.

The response of the network is shown in Figure 14.3. The open circles represent the square-wave input signal $p(t)$. The dots represent the network response $a(t)$. For this dynamic network, the response at any time point depends on the previous three input values. If the input is constant, the output will become constant after three time steps. This type of linear network is called a Finite Impulse Response (FIR) filter.

FIR

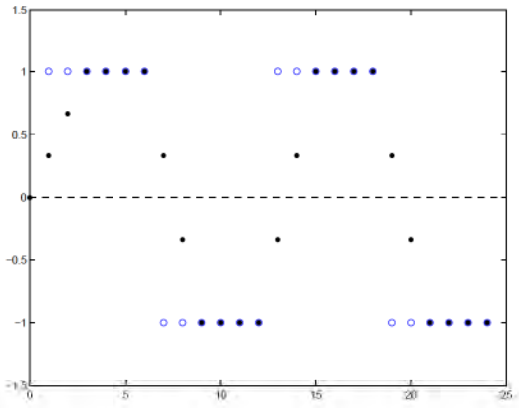


Figure 14.3 Response of ADALINE Filter Network

This dynamic network has memory. Its response at any given time will depend not only on the current input, but on the history of the input sequence. If the network does not have any feedback connections, then only a finite amount of history will affect the response. In the next example, we will consider a network that has an infinite memory.



To experiment with the finite impulse response example, use the Neural Network Design Demonstration Finite Impulse Response Network (`nnd14fir`).



Now consider another simple linear dynamic network, but one that has a recurrent connection. The network in Figure 14.4 is a recurrent dynamic network. The equation of operation of the network is

$$\begin{aligned} \mathbf{a}^1(t) &= \mathbf{n}^1(t) = \mathbf{LW}^{1,1}(1)\mathbf{a}^1(t-1) + \mathbf{IW}^{1,1}(0)\mathbf{p}^1(t) \\ &= lw_{1,1}(1)a(t-1) + iw_{1,1}p(t) \end{aligned} \tag{14.5}$$

where, in the last line, we have left off the superscripts, since there is only one neuron and one layer in the network. To demonstrate the operation of this network, we will set the weight values to

$$lw_{1,1}(1) = \frac{1}{2} \text{ and } iw_{1,1} = \frac{1}{2}. \tag{14.6}$$

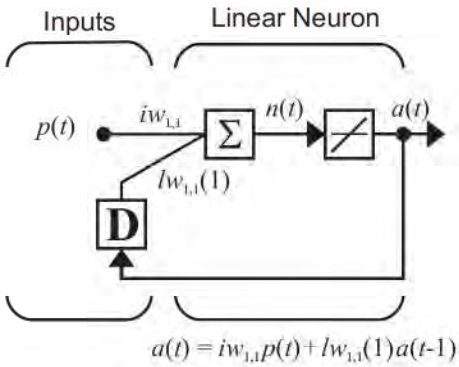


Figure 14.4 Recurrent Linear Neuron

The response of this network to the square wave input is shown in Figure 14.5. The network responds exponentially to a change in the input sequence. Unlike the FIR filter network of Figure 14.2, the exact response of the network at any given time is a function of the infinite history of inputs to the network.

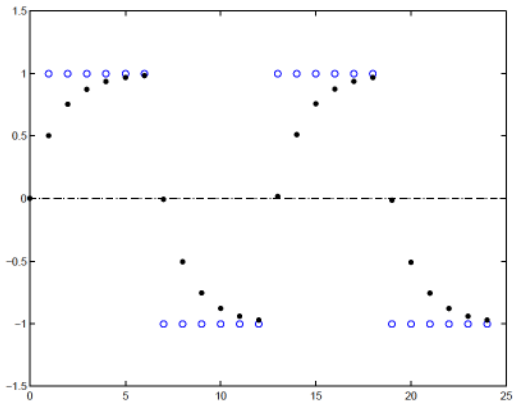


Figure 14.5 Recurrent Neuron Response



To experiment with this infinite impulse response example, use the Neural Network Design Demonstration Infinite Impulse Response Network (`nnd14iir`).

Compare the dynamic networks of the previous two examples with the static, two-layer perceptron of Figure 11.4. Static networks can be trained to approximate static functions, like $\sin(p)$, where the output can be computed directly from the current input. Dynamic networks, on the other hand,

can be trained to approximate dynamic systems, such as robot arms, aircraft, biological processes and economic systems, where the current system output depends on a history of previous inputs and outputs. Because dynamic systems are more complex than static functions, we expect that the training process for dynamic networks will be more challenging than static network training.

In the following section, we will discuss the computation of gradients for the training of dynamic networks. For static networks, these gradients were computed using the standard backpropagation algorithm. For dynamic networks, the backpropagation algorithm must be modified.

Principles of Dynamic Learning



Before we get into the details of training dynamic networks, let's first investigate a simple example. Consider again the recurrent network of Figure 14.4. Suppose that we want to train the network using steepest descent. The first step is to compute the gradient of the performance function. For this example we will use sum squared error:

$$F(\mathbf{x}) = \sum_{t=1}^{Q} e^2(t) = \sum_{t=1}^{Q} (t(t) - a(t))^2 . \tag{14.7}$$

The two elements of the gradient will be

$$\frac{\partial F(\mathbf{x})}{\partial lw_{1,1}(1)} = \sum_{t=1}^{Q} \frac{\partial e^2(t)}{\partial lw_{1,1}(1)} = -2 \sum_{t=1}^{Q} e(t) \frac{\partial a(t)}{\partial lw_{1,1}(1)}, \tag{14.8}$$

$$\frac{\partial F(\mathbf{x})}{\partial iw_{1,1}} = \sum_{t=1}^{Q} \frac{\partial e^2(t)}{\partial iw_{1,1}} = -2 \sum_{t=1}^{Q} e(t) \frac{\partial a(t)}{\partial iw_{1,1}} \tag{14.9}$$

The key terms in these equations are the derivatives of the network output with respect to the weights:

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} \text{ and } \frac{\partial a(t)}{\partial iw_{1,1}} . \tag{14.10}$$

If we had a static network, then these terms would be very easy to compute. They would correspond to $a(t-1)$ and $p(t)$, respectively. However, for recurrent networks, the weights have two effects on the network output. The first is the direct effect, which is also seen in the corresponding static network. The second is an indirect effect, caused by the fact that one of the network inputs is a previous network output. Let's compute the derivatives of the network output, in order to demonstrate these two effects.

The equation of operation of the network is

$$a(t) = lw_{1,1}(1)a(t-1) + iw_{1,1}p(t). \tag{14.11}$$

We can compute the terms in Eq. (14.10) by taking the derivatives of Eq. (14.11):

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} = a(t-1) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial lw_{1,1}(1)}, \tag{14.12}$$

$$\frac{\partial a(t)}{\partial iw_{1,1}} = p(t) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial iw_{1,1}}. \tag{14.13}$$

The first term in each of these equations represents the direct effect that each weight has on the network output. The second term represents the indirect effect. Note that unlike the gradient computation for static networks, the derivative at each time point depends on the derivative at previous time points (or at future time points, as we will see later).

The following figures illustrate the dynamic derivatives. In Figure 14.6 a) we see the total derivatives $\partial a(t)/\partial iw_{1,1}$ and also the static portions of the derivatives. Note that if we consider only the static portion, we will underestimate the effect of a change in the weight. In Figure 14.6 b) we see the original response of the network (which was also shown in Figure 14.5) and a new response, in which $iw_{1,1}$ is increased from 0.5 to 0.6. By comparing the two parts of Figure 14.6, we can see how the derivative indicates the effect on the network response of a change in the weight $iw_{1,1}$.

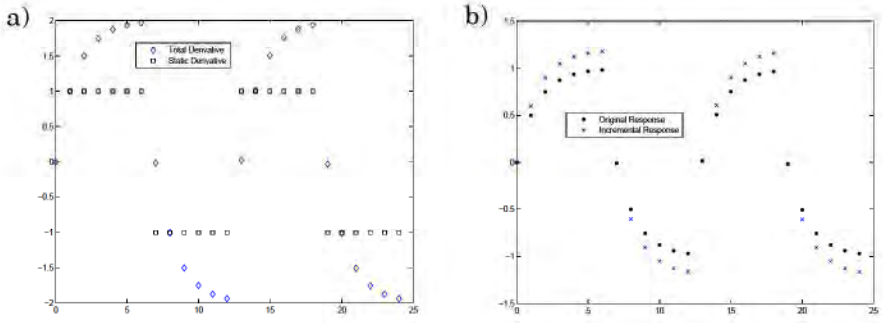


Figure 14.6 Derivatives for $iw_{1,1}$ and Response of Network in Figure 14.4

In Figure 14.7 we see similar results for the weight $lw_{1,1}(1)$. The key ideas to get from this example are: 1) the derivatives have static and dynamic components, and 2) the dynamic component depends on other time points.



To experiment with dynamic derivatives, use the Neural Network Design Demonstration Dynamic Derivatives (**nnd14dynd**).

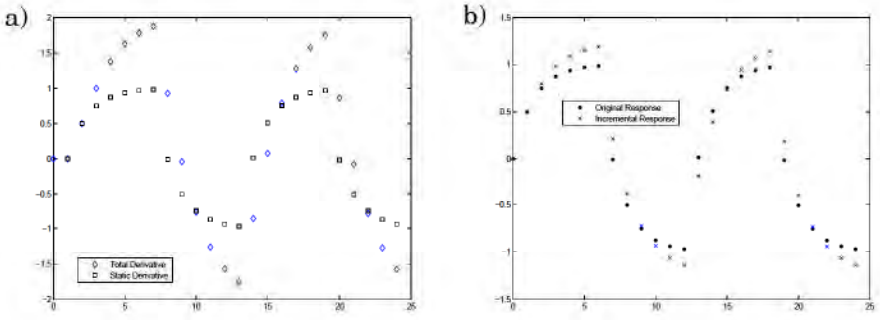


Figure 14.7 Derivatives for $lw_{1,1}(1)$ and Response of Network in Figure 14.4

Having made this initial investigation of a single-neuron network, let's consider the slightly more complex dynamic network that is shown in Figure 14.8. It consists of a static multilayer network with a single feedback loop added from the output of the network to the input of the network through a single delay. In this figure, the vector $\mathbf{x}$ represents all of the network parameters (weights and biases), and the vector $\mathbf{a}(t)$ represents the output of the multilayer network at time step t. This network will help us demonstrate the key steps of dynamic training.

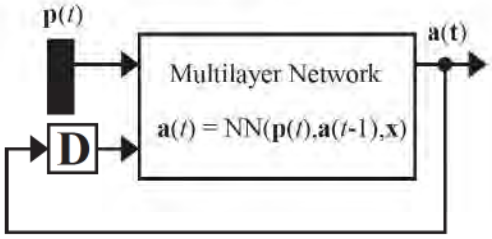


Figure 14.8 Simple Dynamic Network

As with a standard multilayer network, we want to adjust the weights and biases of the network to minimize the performance index, $F(\mathbf{x})$, which is normally chosen to be the mean squared error. In Chapter 11, we derived the backpropagation algorithm for computing the gradient of $F(\mathbf{x})$, which we could then use with any of the optimization methods from Chapter 12 to minimize $F(\mathbf{x})$. With dynamic networks, we need to modify the standard backpropagation algorithm. There are two different approaches to this problem. They both use the chain rule, but are implemented in different ways:

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)}, \tag{14.14}$$

or

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)}. \tag{14.15}$$

where the superscript e indicates an explicit derivative, not accounting for indirect effects through time. The explicit derivatives can be obtained with the standard backpropagation algorithm of Chapter 11. To find the complete derivatives that are required in Eq. (14.14) and Eq. (14.15), we need the additional equations:

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}^T(t-1)} \times \frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}^T} \tag{14.16}$$

and

$$\frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}^T(t)} \times \frac{\partial F}{\partial \mathbf{a}(t+1)}. \tag{14.17}$$

RTRL

Eq. (14.14) and Eq. (14.16) make up the real-time recurrent learning (RTRL) algorithm. Note that the key term is

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T}, \tag{14.18}$$

which must be propagated forward through time.

BPTT

Eq. (14.15) and Eq. (14.17) make up the backpropagation-through-time (BPTT) algorithm. Here the key term is

$$\frac{\partial F}{\partial \mathbf{a}(t)} \tag{14.19}$$

which must be propagated backward through time.

In general, the RTRL algorithm requires somewhat more computation than the BPTT algorithm to compute the gradient. However, the BPTT algorithm cannot be conveniently implemented in real time, since the outputs must be computed for all time steps, and then the derivatives must be backpropagated back to the initial time point. The RTRL algorithm is well suited for real time implementation, since the derivatives can be calculated at each time step. (For Jacobian calculations, which are needed for Levenberg-Marquardt algorithms, the RTRL algorithm is often more efficient than the BPTT algorithm. See [DeHa07].)

Dynamic Backpropagation

In this section, we will develop general RTRL and BPTT algorithms for dynamic networks represented in the LDDN framework. This development will involve generalizing Eq. (14.14) through Eq. (14.17).

Preliminary Definitions

Input Layer

Output Layer

In order to simplify the description of the training algorithm, some layers of the LDDN will be assigned as network outputs, and some will be assigned as network inputs. A layer is an *input layer* if it has an input weight, or if it contains any delays with any of its weight matrices. A layer is an *output layer* if its output will be compared to a target during training, or if it is connected to an input layer through a matrix that has any delays associated with it.

For example, the LDDN shown in Figure 14.1 has two output layers (1 and 3) and two input layers (1 and 2). For this network the simulation order is 1-2-3, and the backpropagation order is 3-2-1. As an aid in later derivations, we will define U as the set of all output layer numbers and X as the set of all input layer numbers. For the LDDN in Figure 14.1, U={1,3} and X={1,2}.

The general equations for simulating an arbitrary LDDN network are given in Eq. (14.1) and Eq. (14.2). At each time point, these equations are iterated forward through the layers, as m is incremented through the simulation order. Time is then incremented from $t = 1$ to $t = Q$.

Real Time Recurrent Learning

In this subsection we will generalize the RTRL algorithm, given in Eq. (14.14) and Eq. (14.16), for LDDN networks. This development will follow in many respects the development of the backpropagation algorithm for static multilayer networks in Chapter 11. You may want to quickly review that material before proceeding.

Eq. (14.14)

The first step in developing the RTRL algorithm is to generalize Eq. (14.14). For the general LDDN network, we can calculate the terms of the gradient by using the chain rule, as in

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right]. \tag{14.20}$$

If we compare this equation with Eq. (14.14), we notice that in addition to each time step, we also have a term in the sum for each output layer. However, if the performance index $F(\mathbf{x})$ is not explicitly a function of a specific output $\mathbf{a}^u(t)$, then that explicit derivative will be zero.

Eq. (14.16)

The next step of the development of the RTRL algorithm is the generalization of Eq. (14.16). Again, we use the chain rule:

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u'}} \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}. \quad (14.21)$$

In Eq. (14.16) we only had one delay in the system. Now we need to account for each output and also for the number of times each output is delayed before it is input to another layer. That is the reason for the first two summations in Eq. (14.21). These equations must be updated forward in time, as t is varied from 1 to Q. The terms

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \quad (14.22)$$

are generally set to zero for $t \le 0$.

To implement Eq. (14.21), we need to compute the terms

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T}. \quad (14.23)$$

To find the second term on the right, we can use

$$\begin{aligned} n_k^x(t) = &\sum_{l \in L_s^l} \sum_{d' \in DL_{x,l}} \left[\sum_{i=1}^{S^l} lw_{k,i}^{x,l}(d') a_i^l(t-d') \right] \\ &+ \sum_{l \in I_x} \sum_{d' \in DI_{x,l}} \left[\sum_{i=1}^{R^l} iw_{k,i}^{x,l}(d') p_i^l(t-d') \right] + b_k^x \end{aligned} \quad (14.24)$$

(Compare with Eq. (11.20).) We can now write

$$\frac{\partial^e n_k^x(t)}{\partial a_j^{u'}(t-d)} = lw_{k,j}^{x,u'}(d). \quad (14.25)$$

If we define the following sensitivity term

$$s_{k,i}^{u,m}(t) \equiv \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}, \quad (14.26)$$

which can be used to make up the following matrix

$$\mathbf{S}^{u,m}(t) = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^m(t)^T} = \begin{bmatrix} s_{1,1}^{u,m}(t) & s_{1,2}^{u,m}(t) & \dots & s_{1,S_m}^{u,m}(t) \\ s_{2,1}^{u,m}(t) & s_{2,2}^{u,m}(t) & \dots & s_{1,S_m}^{u,m}(t) \\ \vdots & \vdots & & \vdots \\ s_{S_u,1}^{u,m}(t) & s_{S_u,2}^{u,m}(t) & \dots & s_{S_u,S_m}^{u,m}(t) \end{bmatrix}, \tag{14.27}$$

then we can write Eq. (14.23) as

$$\left[\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T}\right]_{i,j} = \sum_{k=1}^{S^x} s_{i,k}^{u,x}(t+d) \times lw_{k,j}^{x,u'}(d), \tag{14.28}$$

or in matrix form

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{n}^x(t)^T} \times \frac{\partial^e \mathbf{n}^x(t)}{\partial \mathbf{a}^{u'}(t-d)^T} = \mathbf{S}^{u,x}(t) \times \mathbf{LW}^{x,u'}(d). \tag{14.29}$$

Therefore Eq. (14.21) can be written

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} \mathbf{S}^{u,x}(t) \times \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T} \tag{14.30}$$

Many of the terms in the summation on the right hand side of Eq. (14.30) will be zero and will not have to be computed. To take advantage of these efficiencies, we introduce some indicator sets. They are sets that tell us for which layers the weights and the sensitivities are nonzero.

The first type of indicator set contains all of the output layers that connect to a specified layer x (which will always be an input layer) with at least some nonzero delay:

$$E_{LW}^U(x) = \{u \in U \ni \exists(\mathbf{LW}^{x,u}(d) \neq 0, d \neq 0)\}, \tag{14.31}$$

where $\ni$ means "such that," and $\exists$ means "there exists."

The second type of indicator set contains the input layers that have a nonzero sensitivity with a specified layer u:

$$E_S^X(u) = \{x \in X \ni \exists(\mathbf{S}^{u,x} \neq 0)\}. \tag{14.32}$$

(When $\mathbf{S}^{u,x}$ is nonzero, there is a static connection from layer x to ouput layer u.) The third type of indicator set contains the layers that have a nonzero sensitivity with a specified layer u:

$$E_S(u) = \{x \ni \exists(\mathbf{S}^{u,x} \neq 0)\}. \tag{14.33}$$

The difference between $E_S^X(u)$ and $E_S(u)$ is that $E_S^X(u)$ contains only input layers. $E_S(u)$ will not be needed in the simplification of Eq. (14.30), but it will be used for the calculation of sensitivities in Eq. (14.38).

Using Eq. (14.31) and Eq. (14.32), we can rearrange the order of the summations in Eq. (14.30) and sum only over nonzero terms:

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}. \quad (14.34)$$

Eq. (14.34) makes up the generalization of Eq. (14.16) for the LDDN network. It remains to compute the sensitivity matrices $\mathbf{S}^{u,m}(t)$ and the explicit derivatives $\partial^e \mathbf{a}^u(t)/\partial w$, which are described in the next two subsections.

Sensitivities

In order to compute the elements of the sensitivity matrix, we use a form of standard static backpropagation. The sensitivities at the outputs of the network can be computed as

$$s_{k,i}^{u,u}(t) = \frac{\partial^e a_k^u(t)}{\partial n_i^u(t)} = \begin{cases} \dot{f}^u(n_i^u(t)) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases}, \quad u \in U, \quad (14.35)$$

or, in matrix form,

$$\mathbf{S}^{u,u}(t) = \dot{\mathbf{F}}^u(\mathbf{n}^u(t)), \quad (14.36)$$

where $\dot{\mathbf{F}}^u(\mathbf{n}^u(t))$ is defined as

$$\dot{\mathbf{F}}^u(\mathbf{n}^u(t)) = \begin{bmatrix} \dot{f}^u(n_1^u(t)) & 0 & \dots & 0 \\ 0 & \dot{f}^u(n_2^u(t)) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^u(n_{S^u}^u(t)) \end{bmatrix} \quad (14.37)$$

(see also Eq. (11.34)). The matrices $\mathbf{S}^{u,m}(t)$ can be computed by backpropagating through the network, from each network output, using

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t) \mathbf{LW}^{l,m}(0) \right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t)), \quad u \in U, \quad (14.38)$$

where m is decremented from u through the backpropagation order, and L_m^b is the set of indices of layers that are directly connected backwards to layer m (or to which layer m connects forward) and that contain no delays in the connection. The backpropagation step given in Eq. (14.38) is essentially the same as that given in Eq. (11.45), but it is generalized to allow for arbitrary connections between layers.

Explicit Derivatives

We also need to compute the explicit derivatives

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T}. \tag{14.39}$$

Using the chain rule of calculus, we can derive the following expansion of Eq. (14.39) for input weights:

$$\frac{\partial^e a_k^u(t)}{\partial iw_{i,j}^{m,l}(d)} = \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \times \frac{\partial^e n_i^m(t)}{\partial iw_{i,j}^{m,l}(d)} = s_{k,i}^{u,m}(t) \times p_j^l(t-d). \tag{14.40}$$

In vector form we can write

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial iw_{i,j}^{m,l}(d)} = \mathbf{s}_i^{u,m}(t) \times p_j^l(t-d). \tag{14.41}$$

In matrix form we have

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t), \tag{14.42}$$

and in a similar way we can derive the derivatives for layer weights and biases:

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t), \tag{14.43}$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t), \tag{14.44}$$

where the *vec* operator transforms a matrix into a vector by stacking the columns of the matrix one underneath the other, and $\mathbf{A} \otimes \mathbf{B}$ is the Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ [MaNe99].

The total RTRL algorithm for the LDDN network is summarized in the following pseudo code.

Real-Time Recurrent Learning Gradient

Initialize:

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \mathbf{0}, t \le 0, \text{ for all } u \in U,$$

For $t = 1$ to Q,

$U' = \varnothing$, $E_S(u) = \varnothing$ and $E_S^X(u) = \varnothing$ for all $u \in U$.

For m decremented through the BP order

For all $u \in U'$, if $E_S(u) \cap L_m^b \neq \varnothing$

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t) \mathbf{LW}^{l,m}(0) \right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the set $E_S(u)$

if $m \in X$, add m to the set $E_S^X(u)$

EndFor u

If $m \in U$

$$\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the sets U' and $E_S(m)$

if $m \in X$, add m to the set $E_S^X(m)$

EndIf m

EndFor m

For $u \in U$ incremented through the simulation order

For all weights and biases ($\mathbf{x}$ is a vector containing all weights and biases)

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$$

EndFor weights and biases

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}$$

EndFor u

EndFor t

Compute Gradients

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right]$$

Example RTRL Implementations (FIR and IIR)



To demonstrate the RTRL algorithm, consider again the feedforward dynamic network of Figure 14.2. The equation of operation of this network is

$$a(t) = n(t) = iw_{1,1}(0)p(t) + iw_{1,1}(1)p(t-1) + iw_{1,1}(2)p(t-2).$$

The architecture of the network is defined by

$$U = \{1\}, X = \{1\}, I_1 = \{1\}, DI_{1,1} = \{0, 1, 2\}, L_1^f = \varnothing, E_{LW}^U(1) = \varnothing.$$

We will choose the following standard performance function with three time points:

$$F = \sum_{t=1}^{Q} (t(t) - a(t))^2 = \sum_{t=1}^{3} e^2(t) = e^2(1) + e^2(2) + e^2(3),$$

with the following inputs and targets:

$$\{p(1), t(1)\}, \{p(2), t(2)\}, \{p(3), t(3)\}.$$

The RTRL algorithm begins with some initialization:

$$U' = \varnothing, E_S(1) = \varnothing, E_S^X(1) = \varnothing.$$

In addition, the initial conditions for the delays, $p(0), p(-1)$, must be provided.

The network response is then computed for the first time step:

$$a(1) = n(1) = iw_{1,1}(0)p(1) + iw_{1,1}(1)p(0) + iw_{1,1}(2)p(-1)$$

Because the RTRL algorithm proceeds forward through time, we can immediately compute the derivatives for the first time step. We will see in the next section that the BPTT algorithm, which proceeds backward through time, will require that we proceed through all of the time points before we can compute the derivatives.

From the preceding pseudo-code, the first step in the derivative calculation will be

$$\mathbf{S}^{1,1}(1) = \dot{\mathbf{F}}^1(\mathbf{n}^1(1)) = 1,$$

since the transfer function is linear. We also update the following sets:

$$E_S^X(1) = \{1\}, E_S(1) = \{1\}.$$

The next step is the computation of the explicit derivatives from Eq. (14.42):

$$\frac{\partial^e \mathbf{a}^1(1)}{\partial vec(\mathbf{IW}^{1,1}(0))^T} = \frac{\partial^e a(1)}{\partial iw_{1,1}(0)} = [\mathbf{p}^1(1)]^T \otimes \mathbf{S}^{1,1}(t) = p(1),$$

$$\frac{\partial^e \mathbf{a}^1(1)}{\partial vec(\mathbf{IW}^{1,1}(1))^T} = \frac{\partial^e a(1)}{\partial iw_{1,1}(1)} = [\mathbf{p}^1(0)]^T \otimes \mathbf{S}^{1,1}(t) = p(0),$$

$$\frac{\partial^e \mathbf{a}^1(1)}{\partial vec(\mathbf{IW}^{1,1}(2))^T} = \frac{\partial^e a(1)}{\partial iw_{1,1}(2)} = [\mathbf{p}^1(-1)]^T \otimes \mathbf{S}^{1,1}(t) = p(-1).$$

The next step would be to compute the total derivative, using Eq. (14.34). However, since $E_{LW}^{U}(1) = \varnothing$, the total derivatives are equal to the explicit derivatives.

All of the above steps would be repeated for each time point, and then the final step is to compute the derivatives of the performance index with respect to the weights, using Eq. (14.20):

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right] = \sum_{t=1}^{3} \left[\left[\frac{\partial \mathbf{a}^1(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^1(t)} \right].$$

If we break this down for each weight, we have

$$\frac{\partial F}{\partial iw_{1,1}(0)} = p(1)(-2e(1)) + p(2)(-2e(2)) + p(3)(-2e(3)),$$

$$\frac{\partial F}{\partial iw_{1,1}(1)} = p(0)(-2e(1)) + p(1)(-2e(2)) + p(2)(-2e(3)),$$

$$\frac{\partial F}{\partial iw_{1,1}(2)} = p(-1)(-2e(1)) + p(0)(-2e(2)) + p(1)(-2e(3)).$$

We can then use this gradient in any of our standard optimization algorithms from Chapters 9 and 12. Note that if we use steepest descent, this result is a batch form of the LMS algorithm (see Eq. (10.33)).



Let's now do an example using a recurrent neural network. Consider again the simple recurrent network in Figure 14.4. From Eq. (14.5), the equation of operation of this network is

$$a(t) = lw_{1,1}(1)a(t-1) + iw_{1,1}p(t)$$

The architecture of the network is defined by

$$U = \{1\}, X = \{1\}, I_1 = \{1\}, DI_{1,1} = \{0\},$$

$$DL_{1,1} = \{1\}, L_1^f = \{1\}, E_{LW}^U(1) = \{1\}.$$

We will choose the same performance function as the previous example:

$$F = \sum_{t=1}^{Q} (t(t) - a(t))^2 = \sum_{t=1}^{3} e^2(t) = e^2(1) + e^2(2) + e^2(3),$$

with the following inputs and targets:

$$\{p(1), t(1)\}, \{p(2), t(2)\}, \{p(3), t(3)\}.$$

We initialize with

$$U' = \varnothing, E_S(1) = \varnothing, E_S^X(1) = \varnothing.$$

In addition, the initial condition for the delay, $a(0)$, and the initial derivatives

$$\frac{\partial a(0)}{\partial iw_{1,1}} \text{ and } \frac{\partial a(0)}{\partial lw_{1,1}(1)}$$

must be provided. (The initial derivatives are usually set to zero.)

The network response is then computed for the first time step:

$$a(1) = lw_{1,1}(1)a(0) + iw_{1,1}p(1)$$

The derivative calculation begins with

$$\mathbf{S}^{1,1}(1) = \dot{\mathbf{F}}^1(\mathbf{n}^1(1)) = 1,$$

since the transfer function is linear. We also update the following sets:

$$E_S^X(1) = \{1\}, E_S(1) = \{1\}.$$

The next step is the computation of the explicit derivatives:

$$\frac{\partial^e \mathbf{a}^1(1)}{\partial vec(\mathbf{IW}^{1,1}(0))^T} = \frac{\partial^e a(1)}{\partial iw_{1,1}} = [\mathbf{p}^1(1)]^T \otimes \mathbf{S}^{1,1}(1) = p(1),$$

$$\frac{\partial^e \mathbf{a}^1(1)}{\partial vec(\mathbf{LW}^{1,1}(1))^T} = \frac{\partial^e a(1)}{\partial lw_{1,1}(1)} = [\mathbf{a}^1(0)]^T \otimes \mathbf{S}^{1,1}(1) = a(0).$$

The next step is to compute the total derivative, using Eq. (14.34):

$$\frac{\partial \mathbf{a}^1(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^1(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{1,1}(t)\mathbf{LW}^{1,1}(1)\frac{\partial \mathbf{a}^1(t-1)}{\partial \mathbf{x}^T}. \tag{14.45}$$

Replicating this formula for each of our weights for this network, for $t = 1$, we have

$$\frac{\partial a(1)}{\partial iw_{1,1}} = p(1) + lw_{1,1}(1)\frac{\partial a(0)}{\partial iw_{1,1}} = p(1),$$

$$\frac{\partial a(1)}{\partial lw_{1,1}(1)} = a(0) + lw_{1,1}(1)\frac{\partial a(0)}{\partial lw_{1,1}(1)} = a(0).$$

Note that unlike the corresponding equation in the previous example, these equations are recursive. The derivative at the current time depends on the derivative at the previous time. (Note that the two initial derivatives on the right side of this equation would normally be set to zero, but at the next time step they would be nonzero.) As we mentioned earlier, the weights in a recurrent network have two different effects on the network output. The first is the direct effect, which is represented by the explicit derivative in Eq. (14.45). The second is an indirect effect, since one of the inputs to the network is a previous output, which is also a function of the weights. This effect causes the second term in Eq. (14.45).

All of the above steps would be repeated for each time point:

$$\frac{\partial^e a(2)}{\partial iw_{1,1}} = p(2), \quad \frac{\partial^e a(2)}{\partial lw_{1,1}(1)} = a(1),$$

$$\frac{\partial a(2)}{\partial iw_{1,1}} = p(2) + lw_{1,1}(1)\frac{\partial a(1)}{\partial iw_{1,1}} = p(2) + lw_{1,1}(1)p(1),$$

$$\frac{\partial a(2)}{\partial lw_{1,1}(1)} = a(1) + lw_{1,1}(1)\frac{\partial a(1)}{\partial lw_{1,1}(1)} = a(1) + lw_{1,1}(1)a(0),$$

$$\frac{\partial^e a(3)}{\partial iw_{1,1}} = p(3), \quad \frac{\partial^e a(3)}{\partial lw_{1,1}(1)} = a(2),$$

$$\frac{\partial a(3)}{\partial iw_{1,1}} = p(3) + lw_{1,1}(1)\frac{\partial a(2)}{\partial iw_{1,1}} = p(3) + lw_{1,1}(1)p(2) + (lw_{1,1}(1))^2 p(1),$$

$$\frac{\partial a(3)}{\partial lw_{1,1}(1)} = a(2) + lw_{1,1}(1)\frac{\partial a(2)}{\partial lw_{1,1}(1)} = a(2) + lw_{1,1}(1)a(1) + (lw_{1,1}(1))^2 a(0).$$

The final step is to compute the derivatives of the performance index with respect to the weights, using Eq. (14.20):

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right] = \sum_{t=1}^{3} \left[\left[\frac{\partial \mathbf{a}^1(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^1(t)} \right].$$

If we break this down for each weight, we have

$$\begin{aligned} \frac{\partial F}{\partial iw_{1,1}} &= \frac{\partial a(1)}{\partial iw_{1,1}}(-2e(1)) + \frac{\partial a(2)}{\partial iw_{1,1}}(-2e(2)) + \frac{\partial a(3)}{\partial iw_{1,1}}(-2e(3)) \\ &= -2e(1)[p(1)] - 2e(2)[p(2) + lw_{1,1}(1)p(1)] \\ &\quad -2e(3)[p(3) + lw_{1,1}(1)p(2) + (lw_{1,1}(1))^2 p(1)] \end{aligned}$$

$$\begin{aligned} \frac{\partial F}{\partial lw_{1,1}(1)} &= \frac{\partial a(1)}{\partial lw_{1,1}(1)}(-2e(1)) + \frac{\partial a(2)}{\partial lw_{1,1}(1)}(-2e(2)) + \frac{\partial a(3)}{\partial lw_{1,1}(1)}(-2e(3)) \\ &= -2e(1)[a(0)] - 2e(2)[a(1) + lw_{1,1}(1)a(0)] \\ &\quad -2e(3)[a(2) + lw_{1,1}(1)a(1) + (lw_{1,1}(1))^2 a(0)] \end{aligned}$$

The expansions that we show in the final two lines of the above equations (and also in some of the previous equations) would not be necessary in practice, since the results would be numerical. We have included them here so that we can compare this result with the BPTT algorithm, which we present next.

Backpropagation-Through-Time

In this section we will generalize the Backpropagation-Through-Time (BPTT) algorithm, given in Eq. (14.15) and Eq. (14.17), for LDDN networks.

Eq. (14.15)

The first step is to generalize Eq. (14.15). For the general LDDN network, we can calculate the terms of the gradient by using the chain rule, as in

$$\frac{\partial F}{\partial lw_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q} \left[\sum_{u \in U} \sum_{k=1}^{S^u} \frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)} \right] \frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} \tag{14.46}$$

(for the layer weights), where u is an output layer, U is the set of all output layers, and S^u is the number of neurons in layer u.

From Eq. (14.24) we can write

$$\frac{\partial^e n_i^m(t)}{\partial lw_{i,j}^{m,l}(d)} = a_j^l(t-d). \tag{14.47}$$

We will also define

$$d_i^m(t) = \sum_{u \in U} \sum_{k=1}^{S^u} \frac{\partial F}{\partial a_k^u(t)} \times \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}. \tag{14.48}$$

The terms of the gradient for the layer weights can then be written

$$\frac{\partial F}{\partial lw_{i,j}^{m,l}(d)} = \sum_{t=1}^{Q} d_i^m(t) a_j^l(t-d), \tag{14.49}$$

If we use the sensitivity term defined in Eq. (14.26),

$$s_{k,i}^{u,m}(t) \equiv \frac{\partial^e a_k^u(t)}{\partial n_i^m(t)}, \tag{14.50}$$

then the elements $d_i^m(t)$ can be written

$$d_i^m(t) = \sum_{u \in U} \sum_{k=1}^{S^u} \frac{\partial F}{\partial a_k^u(t)} \times s_{k,i}^{u,m}(t). \tag{14.51}$$

In matrix form this becomes

$$\mathbf{d}^m(t) = \sum_{u \in U} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)} \tag{14.52}$$

where

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \left[\frac{\partial F}{\partial a_1^u(t)} \; \frac{\partial F}{\partial a_2^u(t)} \; \cdots \; \frac{\partial F}{\partial a_{S_u}^u(t)} \right]^T \tag{14.53}$$

Now the gradient can be written in matrix form.

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{a}^l(t-d)]^T, \tag{14.54}$$

and by similar steps we can find the derivatives for the biases and input weights:

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T, \tag{14.55}$$

$$\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t). \tag{14.56}$$

Eq. (14.54) through Eq. (14.56) make up the generalization of Eq. (14.15) for the LDDN network.

Eq. (14.17)

The next step in the development of the BPTT algorithm is the generalization of Eq. (14.17). Again, we use the chain rule:

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} \left[\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T} \right]^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{14.57}$$

(Many of the terms in these summations will be zero. We will provide a more efficient representation later in this section.) In Eq. (14.17) we only had one delay in the system. Now we need to account for each network output, how that network output is connected back through a network input, and also for the number of times each network output is delayed before it is applied to a network input. That is the reason for the three summations in Eq. (14.57). This equation must be updated backward in time, as t is varied from Q to 1. The terms

$$\frac{\partial F}{\partial \mathbf{a}^{u'}(t)} \tag{14.58}$$

are generally set to zero for $t > Q$.

If we consider the matrix in the brackets on the right side of Eq. (14.57), from Eq. (14.29) we can write

$$\frac{\partial^e \mathbf{a}^{u'}(t+d)}{\partial \mathbf{n}^x(t+d)^T} \times \frac{\partial^e \mathbf{n}^x(t+d)}{\partial \mathbf{a}^u(t)^T} = \mathbf{S}^{u',x}(t+d) \times \mathbf{LW}^{x,u}(d). \tag{14.59}$$

This allows us to write Eq. (14.57) as

$$\frac{\partial F}{\partial \mathbf{a}^{u}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{u}(t)} + \sum_{u' \in U} \sum_{x \in X} \sum_{d \in DL_{x,u}} [\mathbf{S}^{u',x}(t+d) \times \mathbf{LW}^{x,u}(d)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{14.60}$$

Many of the terms in the summation on the right hand side of Eq. (14.60) will be zero and will not have to be computed. In order to provide a more efficient implementation of Eq. (14.60), we define the following indicator sets:

$$E_{LW}^{X}(u) = \{x \in X \ni \exists(\mathbf{LW}^{x,u}(d) \neq 0, d \neq 0)\}, \tag{14.61}$$

$$E_{S}^{U}(x) = \{u \in U \ni \exists(\mathbf{S}^{u,x} \neq 0)\}. \tag{14.62}$$

The first set contains all of the input layers that have a connection from output layer u with at least some nonzero delay. The second set contains output layers that have a nonzero sensitivity with input layer x. When the sensitivity $S^{u,x}$ is nonzero, there is a static connection from input layer x to output layer u.

We can now rearrange the order of the summation in Eq. (14.60) and sum only over the existing terms:

$$\frac{\partial F}{\partial \mathbf{a}^{u}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{u}(t)} + \sum_{x \in E_{LW}^{X}(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^{T} \sum_{u' \in E_{S}^{U}(x)} \mathbf{S}^{u',x}(t+d)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)} \tag{14.63}$$

Summary

The total BPTT algorithm is summarized in the following pseudo code.

Backpropagation-Through-Time Gradient

Initialize:

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \mathbf{0}, t > Q \text{, for all } u \in U,$$

For $t = Q$ to 1,

$U' = \varnothing$, $E_S(u) = \varnothing$, and $E_S^U(u) = \varnothing$ for all $u \in U$.

For m decremented through the BP order

For all $u \in U'$, if $E_S(u) \cap L_m^b \neq \varnothing$

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t) \mathbf{LW}^{l,m}(0) \right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the set $E_S(u)$

add u to the set $E_S^U(m)$

EndFor u

If $m \in U$

$$\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the sets U', $E_S(m)$ and $E_S^U(m)$

EndIf m

EndFor m

For $u \in U$ decremented through the BP order

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{x \in E_{LW}^x(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \sum_{u' \in E_S^U(x)} \mathbf{S}^{u',x}(t+d)^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)}$$

EndFor u

For all layers m

$$\mathbf{d}^m(t) = \sum_{u \in E_S^U(m)} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)}$$

EndFor m

EndFor t

Compute Gradients

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{a}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t)$$

Example BPTT Implementations (FIR and IIR)



To demonstrate the BPTT algorithm, we will use the same example networks that we used for the RTRL algorithm. First, we use the feedforward dynamic network of Figure 14.2. We defined the network architecture on page 14-18.

Before the gradient can be computed using BPTT, the network response must be computed for all time steps:

$$a(1) = n(1) = iw_{1,1}(0)p(1) + iw_{1,1}(1)p(0) + iw_{1,1}(2)p(-1),$$

$$a(2) = n(2) = iw_{1,1}(0)p(2) + iw_{1,1}(1)p(1) + iw_{1,1}(2)p(0),$$

$$a(3) = n(3) = iw_{1,1}(0)p(3) + iw_{1,1}(2)p(0) + iw_{1,1}(2)p(1).$$

The BPTT algorithm begins with some initialization:

$$U' = \varnothing,\ E_S(1) = \varnothing,\ E_S^U(1) = \varnothing.$$

The first step in the derivative calculation will be the sensitivity calculation. For BPTT, we start at the last time point ($t = 3$):

$$\mathbf{S}^{1,1}(3) = \dot{\mathbf{F}}^1(\mathbf{n}^1(3)) = 1,$$

since the transfer function is linear. We also update the following sets:

$$E_S^U(1) = \{1\},\ E_S(1) = \{1\}.$$

The next step is the calculation of the following derivative using Eq. (14.63):

$$\frac{\partial F}{\partial \mathbf{a}^1(3)} = \frac{\partial^e F}{\partial \mathbf{a}^1(3)} = -2e(3).$$

The final step for $t = 3$ is Eq. (14.52):

$$\mathbf{d}^1(3) = [\mathbf{S}^{1,1}(3)]^T \times \frac{\partial F}{\partial \mathbf{a}^1(3)} = -2e(3).$$

We repeat the previous steps for $t = 2$ and $t = 1$, to obtain

$$\mathbf{d}^1(2) = [\mathbf{S}^{1,1}(2)]^T \times \frac{\partial F}{\partial \mathbf{a}^1(2)} = -2e(2),$$

$$\mathbf{d}^1(1) = [\mathbf{S}^{1,1}(1)]^T \times \frac{\partial F}{\partial \mathbf{a}^1(1)} = -2e(1).$$

Now, all time steps are combined in Eq. (14.55):

$$\frac{\partial F}{\partial \mathbf{IW}^{1,1}(0)} = \frac{\partial F}{\partial iw_{1,1}(0)} = \sum_{t=1}^{3} \mathbf{d}^1(t) \times [\mathbf{p}^1(t)]^T = \sum_{t=1}^{3} -2e(t) \times p(t),$$

$$\frac{\partial F}{\partial \mathbf{IW}^{1,1}(1)} = \frac{\partial F}{\partial iw_{1,1}(1)} = \sum_{t=1}^{3} \mathbf{d}^1(t) \times [\mathbf{p}^1(t-1)]^T = \sum_{t=1}^{3} -2e(t) \times p(t-1),$$

$$\frac{\partial F}{\partial \mathbf{IW}^{1,1}(2)} = \frac{\partial F}{\partial iw_{1,1}(2)} = \sum_{t=1}^{3} \mathbf{d}^1(t) \times [\mathbf{p}^1(t-2)]^T = \sum_{t=1}^{3} -2e(t) \times p(t-2).$$

Note that this is the same result we obtained for the RTRL algorithm example on page 14-19. RTRL and BPTT should always produce the same gradient. The only difference is in the implementation.



Let's now use our previous recurrent neural network example of Figure 14.4. We defined the architecture of this network on page 14-19.

Unlike the RTRL algorithm, where initial conditions for the derivatives must be provided, the BPTT algorithm requires final conditions for the derivatives:

$$\frac{\partial a(4)}{\partial iw_{1,1}} \text{ and } \frac{\partial a(4)}{\partial lw_{1,1}(1)},$$

which are normally set to zero.

The network response is then computed for all time steps:

$$a(1) = lw_{1,1}(1)a(0) + iw_{1,1}p(1)$$

$$a(2) = lw_{1,1}(1)a(1) + iw_{1,1}p(2)$$

$$a(3) = lw_{1,1}(1)a(2) + iw_{1,1}p(3)$$

The derivative calculation begins with

$$\mathbf{S}^{1,1}(3) = \dot{\mathbf{F}}^1(\mathbf{n}^1(3)) = 1,$$

since the transfer function is linear. We also update the following sets:

$$E_S^X(1) = \{1\}, E_S(1) = \{1\}.$$

Next we compute the following derivative using Eq. (14.63):

$$\frac{\partial F}{\partial \mathbf{a}^1(t)} = \frac{\partial^e F}{\partial \mathbf{a}^1(t)} + \mathbf{LW}^{1,1}(1)^T \mathbf{S}^{1,1}(t+1)^T \times \frac{\partial F}{\partial \mathbf{a}^1(t+1)}$$

For $t = 3$, we find

$$\frac{\partial F}{\partial \mathbf{a}^1(3)} = \frac{\partial^e F}{\partial \mathbf{a}^1(3)} + lw_{1,1}(1)\mathbf{S}^{1,1}(4)^T \times \cancelto{0}{\frac{\partial F}{\partial \mathbf{a}^1(4)}} = \frac{\partial^e F}{\partial \mathbf{a}^1(3)} = -2e(3)$$

and

$$\mathbf{d}^1(3) = [\mathbf{S}^{1,1}(3)]^T \times \frac{\partial F}{\partial \mathbf{a}^1(3)} = -2e(3)$$

Continuing to $t = 2$,

$$\mathbf{S}^{1,1}(2) = \dot{\mathbf{F}}^1(\mathbf{n}^1(2)) = 1,$$

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{a}^1(2)} &= \frac{\partial^e F}{\partial \mathbf{a}^1(2)} + lw_{1,1}(1)\mathbf{S}^{1,1}(3)^T \times \frac{\partial F}{\partial \mathbf{a}^1(3)} \\ &= -2e(2) + lw_{1,1}(1)(-2e(3))\end{aligned}$$

and

$$\mathbf{d}^1(2) = [\mathbf{S}^{1,1}(2)]^T \times \frac{\partial F}{\partial \mathbf{a}^1(2)} = -2e(2) + lw_{1,1}(1)(-2e(3))$$

Finally, for $t = 1$,

$$\mathbf{S}^{1,1}(1) = \dot{\mathbf{F}}^1(\mathbf{n}^1(1)) = 1,$$

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{a}^1(1)} &= \frac{\partial^e F}{\partial \mathbf{a}^1(1)} + lw_{1,1}(1)\mathbf{S}^{1,1}(2)^T \times \frac{\partial F}{\partial \mathbf{a}^1(2)} \\ &= -2e(1) + lw_{1,1}(1)(-2e(2)) + (lw_{1,1}(1))^2(-2e(3))\end{aligned}$$

and

$$\mathbf{d}^1(1) = [\mathbf{S}^{1,1}(1)]^T \times \frac{\partial F}{\partial \mathbf{a}^1(1)} = -2e(1) + lw_{1,1}(1)(-2e(2)) + (lw_{1,1}(1))^2(-2e(3))$$

Now we can compute the total gradient, using Eq. (14.54) and Eq. (14.55):

$$\frac{\partial F}{\partial \mathbf{LW}^{1,1}(1)} = \frac{\partial F}{\partial lw_{1,1}(1)} = \sum_{t=1}^{3} \mathbf{d}^1(t) \times [\mathbf{a}^1(t-1)]^T$$

$$= a(0)[-2e(1) + lw_{1,1}(1)(-2e(2)) + (lw_{1,1}(1))^2(-2e(3))]$$
$$+ a(1)[-2e(2) + lw_{1,1}(1)(-2e(3))] + a(0)[-2e(3)]$$

$$\frac{\partial F}{\partial \mathbf{IW}^{1,1}(0)} = \frac{\partial F}{\partial iw_{1,1}} = \sum_{t=1}^{3} \mathbf{d}^1(t) \times [\mathbf{p}^1(t)]^T$$

$$= p(1)[-2e(1) + lw_{1,1}(1)(-2e(2)) + (lw_{1,1}(1))^2(-2e(3))]$$
$$+ p(2)[-2e(2) + lw_{1,1}(1)(-2e(3))] + p(3)[-2e(3)]$$

This is the same result that we obtained with the RTRL algorithm on page 14-22.

Summary and Comments on Dynamic Training

The RTRL and BPTT algorithms represent two methods for computing the gradients for dynamic networks. Both algorithms compute the exact gradient, and therefore they produce the same final results. The RTRL algorithm performs the calculations from the first time point forward, which is suitable for on-line (real-time) implementation. The BPTT algorithm starts from the last time point and works backward in time. The BPTT algorithm generally requires fewer computations for the gradient calculation than RTRL, but BPTT usually requires more memory storage.

In addition to the gradient, versions of BPTT and RTRL can be used to compute Jacobian matrices, as are needed in the Levenberg-Marquardt described in Chapter 12. For Jacobian calculations, the RTRL algorithm is generally more efficient that the BPTT algorithm. See [DeHa07] for details.

Once the gradients or Jacobians are computed, many standard optimization algorithms can be used to train the networks. However, training dynamic networks is generally more difficult than training feedforward networks—for a number of reasons. First, a recurrent net can be thought of as a feedforward network, in which the recurrent network is unfolded in time. For example, consider the simple single-layer recurrent network of Figure 14.4. If this network were to be trained over five time steps, we could unfold the network to create 5 layers - one for each time step. If a sigmoid transfer function is used, then if the output of the network is near the saturation point for any time point, the resulting gradient could be quite small.

Another problem in training dynamic networks is the shape of the error surface. It has been shown (see [PhHa13]) that the error surfaces of recurrent networks can have spurious valleys that are not related to the dynam-

ic system that is being approximated. The underlying cause of these valleys is the fact that recurrent networks have the potential for instabilities. For example, the network of Figure 14.4 will be unstable if $lw_{1,1}(1)$ is greater than one in magnitude. However, it is possible, for a particular input sequence, that the network output can be small for a particular value of $lw_{1,1}(1)$ greater than one in magnitude, or for certain combinations of values for $lw_{1,1}(1)$ and $iw_{1,1}$.

Finally, it is sometimes difficult to get adequate training data for dynamic networks. This is because the inputs to some layers will come from tapped delay lines. This means that the elements of the input vector cannot be selected independently, since the time sequence from which they are sampled is generally correlated in time. Unlike static networks, in which the network response depends only on the input to the network at the current time, dynamic network responses depend on the history of the input sequence. The data used to train the network must be representative of all situations for which the network will be used, both in terms of the ranges for each input, but also in terms of the variation of the inputs over time.



To illustrate the training of dynamic networks, consider again the simple recurrent network of Figure 14.4, but let's use a nonlinear sigmoid transfer function, as shown in Figure 14.9.

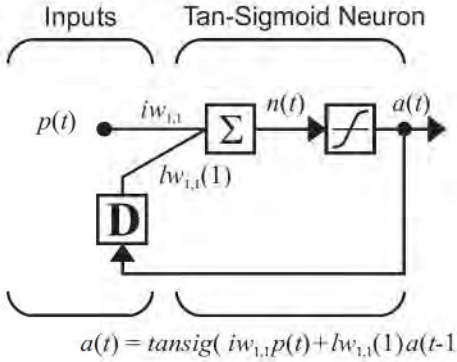


Figure 14.9 Nonlinear Recurrent Network

Recall from Chapter 11 that static multilayer networks can be used to approximate functions. Dynamic networks can be used to approximate dynamic systems. A function maps from one vector space (the domain) to another vector space (the range). A dynamic system maps from one set of time sequences (the input sequences $p(t)$) to another set of time sequences (the output sequences $a(t)$). For example, the network of Figure 14.9 is a dynamic system. It maps from input sequences to output sequences.

In order to simplify our analysis, we will give the network a problem for which we know the optimal solution. The dynamic system we will approximate is the same network, with the following values for the weights:

$$lw_{1,1}(1) = 0.5\,,\ iw_{1,1} = 0.5\,, \tag{14.64}$$

The input sequence that we use to train a dynamic network must be representative of all possible input sequences. Because this network is so simple, it is not difficult to find an appropriate input sequence, but for many practical networks it can be difficult. We will use a standard form of input sequence (called the skyline function), which consists of a series of pulses of varying height and width. The input and target sequences are shown in Figure 14.10. The circles represent the input sequence and the dots represent the target sequence. The targets were created by applying the given input sequence to the network of Figure 14.9, with the weights given by Eq. (14.64).

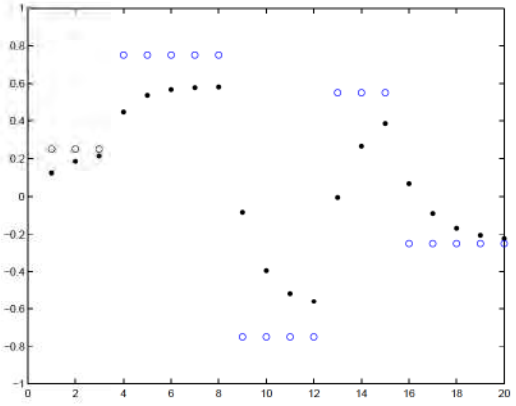


Figure 14.10 Input and Target Sequences

Figure 14.11 shows the squared error performance surface for this problem. Note that as the weight $lw_{1,1}(1)$ becomes greater than one in magnitude, the squared error grows steeply. This effect would be even more prominent, if the length of the training sequence were longer. However, we can also see some narrow valleys in the surface in the regions where $lw_{1,1}(1)$ is greater than one. (This is a very common result, as discussed in [PhHa13]. See Exercise E14.18 to investigate the cause of these valleys.)

The narrow valleys can have an effect on training, since the trajectory can be trapped or misdirected by the spurious valleys. On the left of Figure 14.11 we see a steepest descent path. The path is misdirected at the beginning of the trajectory, because of the narrow valley seen near the bottom of the contour plot.

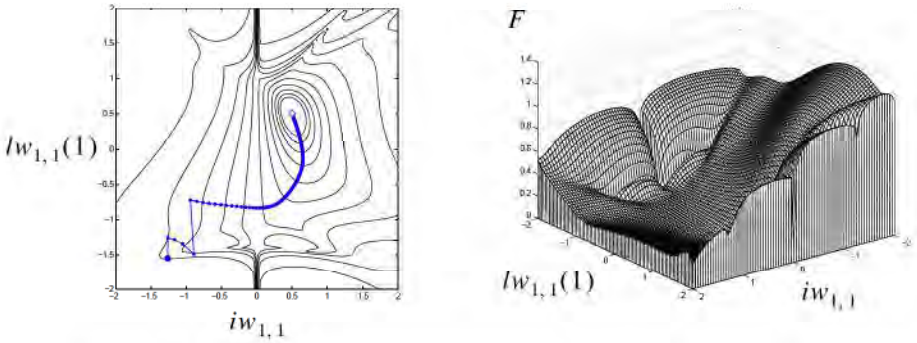


Figure 14.11 Performance Surface and Steepest Descent Trajectory

To experiment with the training of this recurrent network, use the Neural Network Design Demonstration Recurrent Network Training (`nnd14rnt`).

Summary of Results

Real-Time Recurrent Learning Gradient

Initialize:

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \mathbf{0}, t \le 0, \text{ for all } u \in U,$$

For t = 1 to Q,

$U' = \varnothing$, $E_S(u) = \varnothing$ and $E_S^X(u) = \varnothing$ for all $u \in U$.

For m decremented through the BP order

For all $u \in U'$, if $E_S(u) \cap L_m^b \neq \varnothing$

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t) \mathbf{LW}^{l,m}(0) \right] \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the set $E_S(u)$

if $m \in X$, add m to the set $E_S^X(u)$

EndFor u

If $m \in U$

$$\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the sets U' and $E_S(m)$

if $m \in X$, add m to the set $E_S^X(m)$

EndIf m

EndFor m

For $u \in U$ incremented through the simulation order

For all weights and biases ($\mathbf{x}$ is a vector containing all weights and biases)

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$$

EndFor weights and biases

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}$$

EndFor u

EndFor t

Compute Gradients

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)} \right]$$

Backpropagation-Through-Time Gradient

Initialize:

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \mathbf{0}, t > Q \text{, for all } u \in U,$$

For $t = Q$ to 1,

$U' = \varnothing$, $E_S(u) = \varnothing$, and $E_S^U(u) = \varnothing$ for all $u \in U$.

For m decremented through the BP order

For all $u \in U'$, if $E_S(u) \cap L_m^b \neq \varnothing$

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0)\right]\dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the set $E_S(u)$

add u to the set $E_S^U(m)$

EndFor u

If $m \in U$

$$\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the sets U', $E_S(m)$ and $E_S^U(m)$

EndIf m

EndFor m

For $u \in U$ decremented through the BP order

$$\frac{\partial F}{\partial \mathbf{a}^u(t)} = \frac{\partial^e F}{\partial \mathbf{a}^u(t)} + \sum_{x \in E_{LW}^x(u)} \sum_{d \in DL_{x,u}} \mathbf{LW}^{x,u}(d)^T \sum_{u' \in E_S^U(x)} \mathbf{S}^{u',x}(t+d)^T \times \frac{\partial F}{\partial \mathbf{a}^{u'}(t+d)}$$

EndFor u

For all layers m

$$\mathbf{d}^m(t) = \sum_{u \in E_S^U(m)} [\mathbf{S}^{u,m}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^u(t)}$$

EndFor m

EndFor t

Compute Gradients

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{a}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t)$$

Definitions/Notation

$\mathbf{p}^l(t)$ is the l^{th} *input vector* to the network at time t.

$\mathbf{n}^m(t)$ is the *net input* for layer m.

$\mathbf{f}^m(\ )$ is the *transfer function* for layer m.

$\mathbf{a}^m(t)$ is the *output* for layer m.

$\mathbf{IW}^{m,l}$ is the *input weight* between input l and layer m.

$\mathbf{LW}^{m,l}$ is the *layer weight* between layer l and layer m.

$\mathbf{b}^m$ is the *bias vector* for layer m.

$DL_{m,l}$ is the set of all delays in the tapped delay line between Layer l and Layer m.

$DI_{m,l}$ is the set of all delays in the tapped delay line between Input l and Layer m.

I_m is the set of indices of input vectors that connect to layer m.

L_m^f is the set of indices of layers that directly connect *forward* to layer m.

L_m^b is the set of indices of layers that are directly connected backwards to layer m (or to which layer m connects forward) and that contain no delays in the connection.

A layer is an *input layer* if it has an input weight, or if it contains any delays with any of its weight matrices. The set of input layers is X.

A layer is an *output layer* if its output will be compared to a target during training, or if it is connected to an input layer through a matrix that has any delays associated with it. The set of output layers is U.

The *sensitivity* is defined $s_{k,i}^{u,m}(t) \equiv \dfrac{\partial^e a_k^u(t)}{\partial n_i^m(t)}$.

$$E_{LW}^U(x) = \{u \in U \ni \exists(\mathbf{LW}^{x,u}(d) \neq 0, d \neq 0)\}$$

$$E_S^X(u) = \{x \in X \ni \exists(\mathbf{S}^{u,x} \neq 0)\}$$

$$E_S(u) = \{x \ni \exists(\mathbf{S}^{u,x} \neq 0)\}$$

$$E_{LW}^X(u) = \{x \in X \ni \exists(\mathbf{LW}^{x,u}(d) \neq 0, d \neq 0)\}$$

$$E_S^U(x) = \{u \in U \ni \exists(\mathbf{S}^{u,x} \neq 0)\}$$

Solved Problems

P14.1 Before stating the problem, let's first introduce some notation that will allow us to efficiently represent dynamic networks:

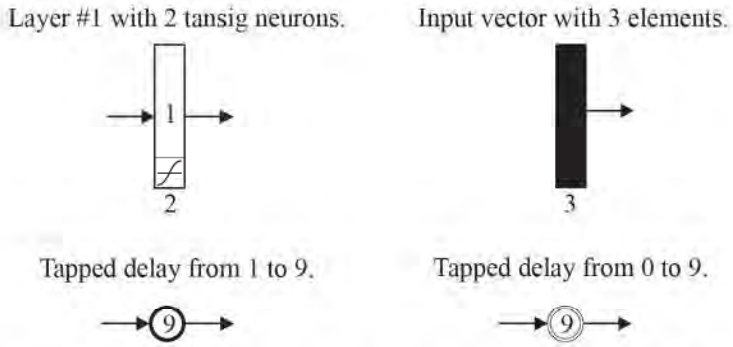


Figure P14.1 Blocks for Dynamic Network Schematics

Using this notation, consider the following network

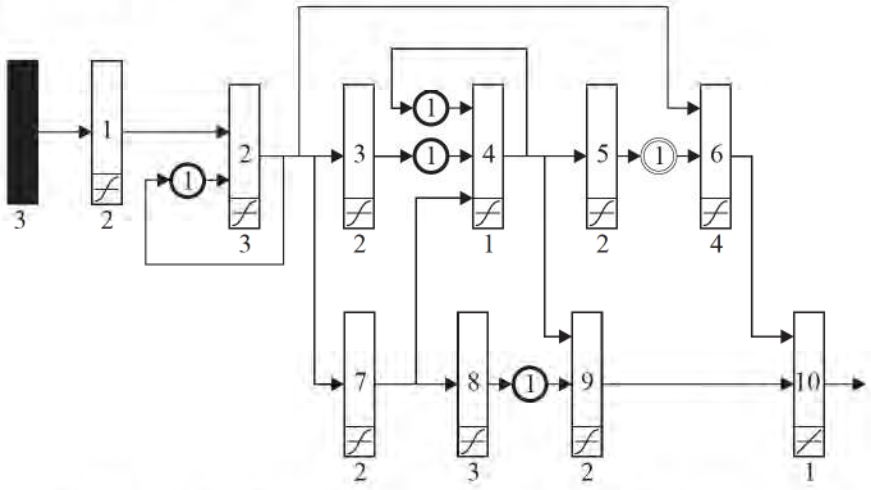


Figure P14.2 Example Dynamic Network for Problem P14.1

Define the network architecture, by showing U, X, I_m, $DI_{m,l}$, $DL_{m,l}$, L_m^f, L_m^b, $E_{LW}^U(x)$, $E_{LW}^X(u)$. Also, select a simulation order and indicate the dimension of each weight matrix.

The input layers have input weights, or have delays with their weight matrices. Therefore $X = \{1, 2, 4, 6, 9\}$. The output layers are compared to targets, or connect to an input layer through a matrix that has any delays. If

we assume that only Layer 10 is compared to a target, then $U = \{2, 3, 4, 5, 8, 10\}$. Since the single input vector connects only to Layer 1, the only nonempty set of inputs will be $I_1 = \{1\}$. For the same reason, there will only be one nonempty set of input delays: $DI_{1,1} = \{0\}$. The connections between layers are defined by

$$L_1^f = \varnothing,\ L_2^f = \{1, 2\},\ L_3^f = \{2\},\ L_4^f = \{3, 4, 7\},\ L_5^f = \{4\},$$

$$L_6^f = \{2, 5\},\ L_7^f = \{2\},\ L_8^f = \{7\},\ L_9^f = \{4, 8\},\ L_{10}^f = \{6, 9\}.$$

$$L_1^b = \{2\},\ L_2^b = \{3, 6, 7\},\ L_3^b = \varnothing,\ L_4^b = \{5, 9\},\ L_5^b = \varnothing,$$

$$L_6^b = \{10\},\ L_7^b = \{4, 8\},\ L_8^b = \varnothing,\ L_9^b = \{10\},\ L_{10}^b = \varnothing.$$

Associated with these connections are the following layer delays

$$DL_{2,1} = \{0\},\ DL_{2,2} = \{1\},\ DL_{3,2} = \{0\},\ DL_{4,3} = \{1\},\ DL_{4,4} = \{1\},$$

$$DL_{4,7} = \{0\},\ DL_{5,4} = \{0\},\ DL_{6,2} = \{0\},\ DL_{6,5} = \{0, 1\},\ DL_{7,2} = \{0\},$$

$$DL_{8,7} = \{0\},\ DL_{9,4} = \{0\},\ DL_{9,8} = \{1\},\ DL_{10,6} = \{0\},\ DL_{10,9} = \{0\}.$$

The layers that have connections from output layers are

$$E_{LW}^U(2) = \{2\},\ E_{LW}^U(4) = \{3, 4\},$$

$$E_{LW}^U(6) = \{5\},\ E_{LW}^U(9) = \{8\}.$$

The layers that connect to input layers are

$$E_{LW}^X(2) = \{2\},\ E_{LW}^X(3) = \{4\},\ E_{LW}^X(4) = \{4\},$$

$$E_{LW}^X(5) = \{6\},\ E_{LW}^X(8) = \{9\}.$$

The simulation order can be chosen to be $\{1, 2, 3, 7, 4, 5, 6, 8, 9, 10\}$. The dimensions of the weight matrices are

$$\mathbf{IW}^{1,1}(0) \Rightarrow 2 \times 3,\ \mathbf{LW}^{2,1}(0) \Rightarrow 3 \times 2,\ \mathbf{LW}^{2,2}(1) \Rightarrow 3 \times 3,\ \mathbf{LW}^{3,2}(0) \Rightarrow 2 \times 3,$$

$$\mathbf{LW}^{4,3}(1) \Rightarrow 1 \times 2,\ \mathbf{LW}^{4,4}(1) \Rightarrow 1 \times 1,\ \mathbf{LW}^{4,7}(0) \Rightarrow 1 \times 2,\ \mathbf{LW}^{5,4}(0) \Rightarrow 2 \times 1,$$

$$\mathbf{LW}^{6,2}(0) \Rightarrow 4 \times 3,\ \mathbf{LW}^{6,5}(1) \Rightarrow 4 \times 2,\ \mathbf{LW}^{7,2}(0) \Rightarrow 2 \times 3,\ \mathbf{LW}^{8,7}(0) \Rightarrow 3 \times 2,$$

$$\mathbf{LW}^{9,4}(0) \Rightarrow 2 \times 1,\ \mathbf{LW}^{9,8}(1) \Rightarrow 2 \times 3,\ \mathbf{LW}^{10,6}(0) \Rightarrow 1 \times 4,\ \mathbf{LW}^{10,9}(d) \Rightarrow 1 \times 2.$$

P14.2 Write out the BPTT equations for the network presented in Problem P14.1.

We will assume that the network response has been computed for all time points, and we will demonstrate the process for one time step, since they all will be similar. We proceed through the layers according to the backpropagation order, which is the reverse of the simulation order: $\{10, 9, 8, 6, 5, 4, 7, 3, 2, 1\}$.

$$\mathbf{S}^{10,10}(t) = \dot{\mathbf{F}}^{10}(\mathbf{n}^{10}(t))$$

$$\frac{\partial F}{\partial \mathbf{a}^{10}(t)} = \frac{\partial^e F}{\partial \mathbf{a}^{10}(t)}$$

$$\mathbf{d}^{10}(t) = [\mathbf{S}^{10,10}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)}$$

$$\mathbf{S}^{10,9}(t) = \mathbf{S}^{10,10}(t)\mathbf{LW}^{10,9}(0)\dot{\mathbf{F}}^{9}(\mathbf{n}^{9}(t))$$

$$\mathbf{d}^{9}(t) = [\mathbf{S}^{10,9}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)}$$

$$\mathbf{S}^{8,8}(t) = \dot{\mathbf{F}}^{8}(\mathbf{n}^{8}(t))$$

$$\frac{\partial F}{\partial \mathbf{a}^{8}(t)} = \frac{\partial^e F}{\partial \mathbf{a}^{8}(t)} + \mathbf{LW}^{9,8}(1)^T \mathbf{S}^{10,9}(t+1)^T \times \frac{\partial F}{\partial \mathbf{a}^{10}(t+1)}$$

$$\mathbf{d}^{8}(t) = [\mathbf{S}^{8,8}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^{8}(t)}$$

$$\mathbf{S}^{10,6}(t) = \mathbf{S}^{10,10}(t)\mathbf{LW}^{10,6}(0)\dot{\mathbf{F}}^{6}(\mathbf{n}^{6}(t))$$

$$\mathbf{d}^{6}(t) = [\mathbf{S}^{10,6}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)}$$

$$\mathbf{S}^{5,5}(t) = \dot{\mathbf{F}}^{5}(\mathbf{n}^{5}(t))$$

$$\frac{\partial F}{\partial \mathbf{a}^{5}(t)} = \frac{\partial^e F}{\partial \mathbf{a}^{5}(t)} + \mathbf{LW}^{6,5}(1)^T \mathbf{S}^{10,6}(t+1)^T \times \frac{\partial F}{\partial \mathbf{a}^{10}(t+1)} + \mathbf{LW}^{6,5}(0)^T \mathbf{S}^{10,6}(t)^T \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)}$$

$$\mathbf{d}^{5}(t) = [\mathbf{S}^{5,5}(t)]^T \times \frac{\partial F}{\partial \mathbf{a}^{5}(t)}$$

$$\mathbf{S}^{10,4}(t) = \mathbf{S}^{10,9}(t)\mathbf{LW}^{9,4}(0)\dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t))$$

$$\mathbf{S}^{5,4}(t) = \mathbf{S}^{5,5}(t)\mathbf{LW}^{5,4}(0)\dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t))$$

$$\mathbf{S}^{4,4}(t) = \dot{\mathbf{F}}^{4}(\mathbf{n}^{4}(t))$$

$$\frac{\partial F}{\partial \mathbf{a}^{4}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{4}(t)} + \mathbf{LW}^{4,4}(1)^{T}\left[\mathbf{S}^{4,4}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t+1)} + \mathbf{S}^{5,4}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t+1)} + \mathbf{S}^{10,4}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t+1)}\right]$$

$$\mathbf{d}^{4}(t) = [\mathbf{S}^{4,4}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t)} + [\mathbf{S}^{5,4}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t)} + [\mathbf{S}^{10,4}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)}$$

$$\mathbf{S}^{10,7}(t) = \mathbf{S}^{10,4}(t)\mathbf{LW}^{4,7}(0)\dot{\mathbf{F}}^{7}(\mathbf{n}^{7}(t))$$

$$\mathbf{S}^{8,7}(t) = \mathbf{S}^{8,8}(t)\mathbf{LW}^{8,7}(0)\dot{\mathbf{F}}^{7}(\mathbf{n}^{7}(t))$$

$$\mathbf{S}^{5,7}(t) = \mathbf{S}^{5,4}(t)\mathbf{LW}^{4,7}(0)\dot{\mathbf{F}}^{7}(\mathbf{n}^{7}(t))$$

$$\mathbf{S}^{4,7}(t) = \mathbf{S}^{4,4}(t)\mathbf{LW}^{4,7}(0)\dot{\mathbf{F}}^{7}(\mathbf{n}^{7}(t))$$

$$\mathbf{d}^{7}(t) = [\mathbf{S}^{10,7}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)} + [\mathbf{S}^{8,7}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{8}(t)} + [\mathbf{S}^{5,7}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t)} + [\mathbf{S}^{4,7}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t)}$$

$$\mathbf{S}^{3,3}(t) = \dot{\mathbf{F}}^{3}(\mathbf{n}^{3}(t))$$

$$\frac{\partial F}{\partial \mathbf{a}^{3}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{3}(t)} + \mathbf{LW}^{4,3}(1)^{T}\left[\mathbf{S}^{4,4}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t+1)} + \mathbf{S}^{5,4}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t+1)} + \mathbf{S}^{10,4}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t+1)}\right]$$

$$\mathbf{d}^{3}(t) = [\mathbf{S}^{3,3}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{3}(t)}$$

$$\mathbf{S}^{10,2}(t) = \mathbf{S}^{10,6}(t)\mathbf{LW}^{6,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t)) + \mathbf{S}^{10,7}(t)\mathbf{LW}^{7,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\mathbf{S}^{8,2}(t) = \mathbf{S}^{8,7}(t)\mathbf{LW}^{7,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\mathbf{S}^{5,2}(t) = \mathbf{S}^{5,7}(t)\mathbf{LW}^{7,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\mathbf{S}^{4,2}(t) = \mathbf{S}^{4,7}(t)\mathbf{LW}^{7,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\mathbf{S}^{3,2}(t) = \mathbf{S}^{3,3}(t)\mathbf{LW}^{3,2}(0)\dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\mathbf{S}^{2,2}(t) = \dot{\mathbf{F}}^{2}(\mathbf{n}^{2}(t))$$

$$\frac{\partial F}{\partial \mathbf{a}^{2}(t)} = \frac{\partial^{e} F}{\partial \mathbf{a}^{2}(t)} + \mathbf{LW}^{2,2}(1)^{T}\left[\mathbf{S}^{2,2}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{2}(t+1)} + \mathbf{S}^{3,2}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{3}(t+1)}\right.$$
$$+ \mathbf{S}^{4,2}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t+1)} + \mathbf{S}^{5,2}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t+1)}$$
$$\left. + \mathbf{S}^{8,2}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{8}(t+1)} + \mathbf{S}^{10,2}(t+1)^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t+1)}\right]$$

$$\mathbf{d}^{2}(t) = [\mathbf{S}^{10,2}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)} + [\mathbf{S}^{8,2}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{8}(t)} + [\mathbf{S}^{5,2}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t)}$$
$$+ [\mathbf{S}^{4,2}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t)} + [\mathbf{S}^{3,2}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{3}(t)} + [\mathbf{S}^{2,2}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{2}(t)}$$

$$\mathbf{S}^{10,1}(t) = \mathbf{S}^{10,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{S}^{8,1}(t) = \mathbf{S}^{8,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{S}^{5,1}(t) = \mathbf{S}^{5,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{S}^{4,1}(t) = \mathbf{S}^{4,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{S}^{3,1}(t) = \mathbf{S}^{3,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{S}^{2,1}(t) = \mathbf{S}^{2,2}(t)\mathbf{LW}^{2,1}(0)\dot{\mathbf{F}}^{1}(\mathbf{n}^{1}(t))$$

$$\mathbf{d}^{1}(t) = [\mathbf{S}^{10,1}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{10}(t)} + [\mathbf{S}^{8,1}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{8}(t)} + [\mathbf{S}^{5,1}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{5}(t)}$$
$$+ [\mathbf{S}^{4,1}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{4}(t)} + [\mathbf{S}^{3,1}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{3}(t)} + [\mathbf{S}^{2,1}(t)]^{T} \times \frac{\partial F}{\partial \mathbf{a}^{2}(t)}$$

After the preceding steps have been repeated for all time points, from the last time point back to the first, then the gradient can be computed as follows:

$$\frac{\partial F}{\partial \mathbf{LW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^{m}(t) \times [\mathbf{a}^{l}(t-d)]^{T}$$

$$\frac{\partial F}{\partial \mathbf{IW}^{m,l}(d)} = \sum_{t=1}^{Q} \mathbf{d}^m(t) \times [\mathbf{p}^l(t-d)]^T$$

$$\frac{\partial F}{\partial \mathbf{b}^m} = \sum_{t=1}^{Q} \mathbf{d}^m(t)$$

P14.3 Write out the RTRL equations for the network presented in Problem P14.1.

As in the previous problem, we will demonstrate the process for one time step, since each step is similar. We will proceed through the layers according to the backpropagation order. The sensitivity matrices $\mathbf{S}^{u,m}(t)$ are computed in the same way for the RTRL algorithm as for the BPTT algorithm, so we won't repeat those steps from Problem P14.2.

The explicit derivative calculations for the input weight will be

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{1,1}(0))^T} = [\mathbf{p}^1(t)]^T \otimes \mathbf{S}^{u,1}(t)$$

For the layer weights and the biases, the explicit derivatives are calculated by

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$$

For the total derivatives, we have

$$\frac{\partial \mathbf{a}^2(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^2(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{2,2}(t)\left[\mathbf{LW}^{2,2}(1) \times \frac{\partial \mathbf{a}^2(t-1)}{\partial \mathbf{x}^T}\right]$$

$$\frac{\partial \mathbf{a}^3(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^3(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{3,2}(t)\left[\mathbf{LW}^{2,2}(1) \times \frac{\partial \mathbf{a}^2(t-1)}{\partial \mathbf{x}^T}\right]$$

$$\frac{\partial \mathbf{a}^4(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^4(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{4,4}(t)\left[\mathbf{LW}^{4,4}(1) \times \frac{\partial \mathbf{a}^4(t-1)}{\partial \mathbf{x}^T} + \mathbf{LW}^{4,3}(1) \times \frac{\partial \mathbf{a}^3(t-1)}{\partial \mathbf{x}^T}\right] + \mathbf{S}^{4,2}(t)\left[\mathbf{LW}^{2,2}(1) \times \frac{\partial \mathbf{a}^2(t-1)}{\partial \mathbf{x}^T}\right]$$

$$\frac{\partial \mathbf{a}^5(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^5(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{5,4}(t)\left[\mathbf{LW}^{4,4}(1) \times \frac{\partial \mathbf{a}^4(t-1)}{\partial \mathbf{x}^T} + \mathbf{LW}^{4,3}(1) \times \frac{\partial \mathbf{a}^3(t-1)}{\partial \mathbf{x}^T}\right] + \mathbf{S}^{5,2}(t)\left[\mathbf{LW}^{2,2}(1) \times \frac{\partial \mathbf{a}^2(t-1)}{\partial \mathbf{x}^T}\right]$$

$$\frac{\partial \mathbf{a}^8(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^8(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{8,2}(t)\left[\mathbf{LW}^{2,2}(1) \times \frac{\partial \mathbf{a}^2(t-1)}{\partial \mathbf{x}^T}\right]$$

$$\frac{\partial \mathbf{a}^{10}(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^{10}(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{10,9}(t)\left[\mathbf{LW}^{9,8}(1) \times \frac{\partial \mathbf{a}^8(t-1)}{\partial \mathbf{x}^T}\right] + \mathbf{S}^{10,6}(t)\left[\mathbf{LW}^{6,5}(0) \times \frac{\partial \mathbf{a}^5(t)}{\partial \mathbf{x}^T} + \mathbf{LW}^{6,5}(1) \times \frac{\partial \mathbf{a}^5(t-1)}{\partial \mathbf{x}^T}\right] + \mathbf{S}^{10,4}(t)\left[\mathbf{LW}^{4,4}(1) \times \frac{\partial \mathbf{a}^4(t-1)}{\partial \mathbf{x}^T} + \mathbf{LW}^{4,3}(1) \times \frac{\partial \mathbf{a}^3(t-1)}{\partial \mathbf{x}^T}\right] + \mathbf{S}^{10,2}(t)\left[\mathbf{LW}^{2,2}(1) \times \frac{\partial \mathbf{a}^2(t-1)}{\partial \mathbf{x}^T}\right]$$

After the above steps are iterated through all time points, we can compute the gradient with

$$\frac{\partial F}{\partial \mathbf{x}^T} = \sum_{t=1}^{Q}\left[\left[\frac{\partial^e F}{\partial \mathbf{a}^2(t)}\right]^T \times \frac{\partial \mathbf{a}^2(t)}{\partial \mathbf{x}^T} + \left[\frac{\partial^e F}{\partial \mathbf{a}^3(t)}\right]^T \times \frac{\partial \mathbf{a}^3(t)}{\partial \mathbf{x}^T} + \left[\frac{\partial^e F}{\partial \mathbf{a}^4(t)}\right]^T \times \frac{\partial \mathbf{a}^4(t)}{\partial \mathbf{x}^T} + \left[\frac{\partial^e F}{\partial \mathbf{a}^5(t)}\right]^T \times \frac{\partial \mathbf{a}^5(t)}{\partial \mathbf{x}^T} + \left[\frac{\partial^e F}{\partial \mathbf{a}^8(t)}\right]^T \times \frac{\partial \mathbf{a}^8(t)}{\partial \mathbf{x}^T} + \left[\frac{\partial^e F}{\partial \mathbf{a}^{10}(t)}\right]^T \times \frac{\partial \mathbf{a}^{10}(t)}{\partial \mathbf{x}^T}\right]$$

P14.4 From the previous problem, show the detail of the calculations involved in the explicit derivative term

$$\frac{\partial^e \mathbf{a}^2(t)}{\partial vec(\mathbf{IW}^{1,1}(0))^T} = [\mathbf{p}^1(t)]^T \otimes \mathbf{S}^{2,1}(t)$$

First, let's display the details of the individual vectors and matrices in this expression:

$$\mathbf{IW}^{1,1}(0) = \begin{bmatrix} iw_{1,1}^{1,1} & iw_{1,2}^{1,1} & iw_{1,3}^{1,1} \\ iw_{2,1}^{1,1} & iw_{2,2}^{1,1} & iw_{2,3}^{1,1} \end{bmatrix}$$

$$vec(\mathbf{IW}^{1,1}(0))^T = \begin{bmatrix} iw_{1,1}^{1,1} & iw_{2,1}^{1,1} & iw_{1,2}^{1,1} & iw_{2,2}^{1,1} & iw_{1,3}^{1,1} & iw_{2,3}^{1,1} \end{bmatrix}$$

$$\mathbf{p}^1(t) = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad \mathbf{S}^{2,1}(t) = \frac{\partial^e \mathbf{a}^2(t)}{\partial \mathbf{n}^1(t)^T} = \begin{bmatrix} s_{1,1}^{2,1} & s_{1,2}^{2,1} \\ s_{2,1}^{2,1} & s_{2,2}^{2,1} \\ s_{3,1}^{2,1} & s_{3,2}^{2,1} \end{bmatrix}$$

$$\frac{\partial^e \mathbf{a}^2(t)}{\partial vec(\mathbf{IW}^{1,1}(0))^T} = \begin{bmatrix} \frac{\partial a_1^2}{\partial iw_{1,1}^{1,1}} & \frac{\partial a_1^2}{\partial iw_{2,1}^{1,1}} & \frac{\partial a_1^2}{\partial iw_{1,2}^{1,1}} & \frac{\partial a_1^2}{\partial iw_{2,2}^{1,1}} & \frac{\partial a_1^2}{\partial iw_{1,3}^{1,1}} & \frac{\partial a_1^2}{\partial iw_{2,3}^{1,1}} \\ \frac{\partial a_2^2}{\partial iw_{1,1}^{1,1}} & \frac{\partial a_2^2}{\partial iw_{2,1}^{1,1}} & \frac{\partial a_2^2}{\partial iw_{1,2}^{1,1}} & \frac{\partial a_2^2}{\partial iw_{2,2}^{1,1}} & \frac{\partial a_2^2}{\partial iw_{1,3}^{1,1}} & \frac{\partial a_2^2}{\partial iw_{2,3}^{1,1}} \\ \frac{\partial a_3^2}{\partial iw_{1,1}^{1,1}} & \frac{\partial a_3^2}{\partial iw_{2,1}^{1,1}} & \frac{\partial a_3^2}{\partial iw_{1,2}^{1,1}} & \frac{\partial a_3^2}{\partial iw_{2,2}^{1,1}} & \frac{\partial a_3^2}{\partial iw_{1,3}^{1,1}} & \frac{\partial a_3^2}{\partial iw_{2,3}^{1,1}} \end{bmatrix}$$

The Kronecker product is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & \dots & a_{1,m}\mathbf{B} \\ \vdots & & \vdots \\ a_{n,1}\mathbf{B} & \dots & a_{n,m}\mathbf{B} \end{bmatrix},$$

therefore

$$[\mathbf{p}^1(t)]^T \otimes \mathbf{S}^{2,1}(t) = \begin{bmatrix} p_1 s_{1,1}^{2,1} & p_1 s_{1,2}^{2,1} & p_2 s_{1,1}^{2,1} & p_2 s_{1,2}^{2,1} & p_3 s_{1,1}^{2,1} & p_3 s_{1,2}^{2,1} \\ p_1 s_{2,1}^{2,1} & p_1 s_{2,2}^{2,1} & p_2 s_{2,1}^{2,1} & p_2 s_{2,2}^{2,1} & p_3 s_{2,1}^{2,1} & p_3 s_{2,2}^{2,1} \\ p_1 s_{3,1}^{2,1} & p_1 s_{3,2}^{2,1} & p_2 s_{3,1}^{2,1} & p_2 s_{3,2}^{2,1} & p_3 s_{3,1}^{2,1} & p_3 s_{3,2}^{2,1} \end{bmatrix}.$$

P14.5 Find the computational complexity for the BPTT and RTRL algorithms applied to the sample network in Figure P14.3 as a function of the number of neurons in Layer 1 (S^1), the number of delays in the tapped delay line (D) and the length of the training sequence (Q).

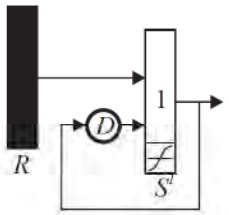


Figure P14.3 Sample Network for Exercise E14.1

The complexity of the BPTT gradient calculation is generally determined by Eq. (14.54). For this network, the most important weights will be $\mathbf{LW}^{1,1}(d)$:

$$\frac{\partial F}{\partial \mathbf{LW}^{1,1}(d)} = \sum_{t=1}^{Q} \mathbf{d}^1(t) \times [\mathbf{a}^1(t-d)]^T,$$

The outer product calculation involves $(S^1)^2$ operations, which must be done for Q time steps and for D delays, so the BPTT gradient calculation is $O[(S^1)^2 DQ]$.

The complexity of the RTRL gradient is based generally on Eq. (14.34). For this sample network, we can consider the equation for $u = 2$:

$$\frac{\partial \mathbf{a}^2(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^2(t)}{\partial \mathbf{x}^T} + \mathbf{S}^{2,1}(t)\left[\sum_{d=1}^{D} \mathbf{LW}^{1,1}(d) \times \frac{\partial \mathbf{a}^1(t-d)}{\partial \mathbf{x}^T}\right]$$

Inside the summation we have a matrix multiplication involving an $S^1 \times S^1$ matrix times an $S^1 \times \{(DS^1 + 3)S^1 + 1\}$ matrix. This multiplication will be $O[(S^1)^4 D]$. It must be done for every d and every t, therefore the RTRL gradient calculations are $O[(S^1)^4 D^2 Q]$. The multiplication by the sensitivity matrix does not change the order of the complexity.

Epilogue

Dynamic networks can be trained using the same optimization procedures that we described in Chapter 12 for static multilayer networks. However, the calculation of the gradient for dynamic networks is more complex than for static networks. There are two basic approaches to the gradient calculation in dynamic networks. One approach, backpropagation through time (BPTT), starts at the last time point, and works backward in time to compute the gradient. The second approach, real-time recurrent learning (RTRL), starts at the first time point, and then works forward through time.

RTRL requires more calculations than BPTT for calculating the gradient, but RTRL allows a convenient framework for on-line implementation. Also, RTRL generally requires less storage than BPTT. For Jacobian calculations, RTRL is often more efficient than the BPTT algorithm.

Chapter 27 presents a real-world case study for using dynamic networks to solve a prediction problem.

Further Reading

[DeHa07] O. De Jesús and M. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp., 2007.

This paper presents a general development of BPTT and RTRL algorithms for gradient and Jacobian calculations. Experimental results are presented that compare the computational complexities of the two algorithms for a variety of network architectures.

[MaNe99] J.R. Magnus and H. Neudecker, *Matrix Differential Calculus*, John Wiley & Sons, Ltd., Chichester, 1999.

This textbook provides a very clear and complete description of matrix theory and matrix differential calculus.

[PhHa13] M. Phan and M. Hagan, "Error Surface of Recurrent Networks," *IEEE Transactions on Neural Networks and Learning Systems*, Vol. 24, No. 11, pp. 1709 - 1721, October, 2013.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also describes some procedures that can be used to improve training of recurrent networks.

[Werb90] P. J. Werbos, "Backpropagation through time: What it is and how to do it," *Proceedings of the IEEE*, vol. 78, pp. 1550–1560, 1990.

The backpropagation through time algorithm is one of the two principal approaches to computing the gradients in recurrent neural network. This paper describes the general framework for backpropagation through time.

[WiZi89] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, pp. 270–280, 1989.

This paper introduced the real-time, recurrent learning algorithm for computing the gradients of dynamic networks. Using this method, the gradients are computed by starting at the first time point, and then working forward through time. The algorithm is suitable for on-line, or real-time, implementation.

Exercises

E14.1 Put the network of Figure 14.1 into the schematic form, which we introduced in Problem P14.1.

E14.2 Consider the network in Figure 14.4, with weight values $iw_{1,1} = 2$ and $lw_{1,1}(1) = 0.5$. If $a(0) = 4$, and $p(1) = 2$, $p(2) = 3$, $p(3) = 2$, find $a(1)$, $a(2)$, $a(3)$.

E14.3 Consider the network in Figure P14.3, with $D = 2$, $S^1 = 1$, $R = 1$, $\mathbf{IW}^{1,1} = [1]$, $\mathbf{LW}^{1,1}(1) = [0.5]$, $\mathbf{LW}^{1,1}(2) = [0.2]$, $\mathbf{a}^1(0) = [2]$ and $\mathbf{a}^1(-1) = [1]$. If $\mathbf{p}^1(1) = [1]$, $\mathbf{p}^1(2) = [2]$ and $\mathbf{p}^1(3) = [-1]$, find $\mathbf{a}^1(1)$, $\mathbf{a}^1(2)$ and $\mathbf{a}^1(3)$.

E14.4 Consider the network in Figure E14.1. Define the network architecture, by showing U, X, I_m, $DI_{m,1}$, $DL_{m,l}$, L_m^f, L_m^b, $E_{LW}^U(x)$, $E_{LW}^X(u)$. Also, select a simulation order and indicate the dimension of each weight matrix.

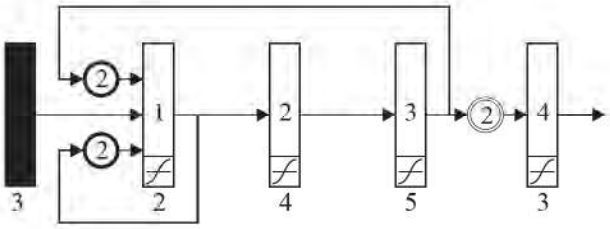


Figure E14.1 Dynamic Network for Exercise E14.4

E14.5 Write out the RTRL equations for the network in Figure E14.2

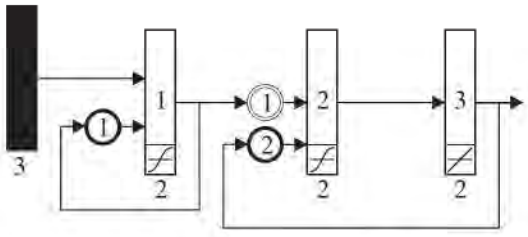


Figure E14.2 Dynamic Network for Exercise E14.5

E14.6 Write out the BPTT equations for the network in Figure E14.2.

E14.7 Write out the equations of operation for the network in Figure E14.3. Assume that all weights have a value of 0.5, and that all biases have a value of 0.

i. Assume that the initial network output is $a(0) = 0.5$, and that the initial network input is $p(1) = 1$. Solve for $a(1)$.

ii. Describe any problems that you would have in simulating this network. Will you be able to apply the BPTT and RTRL algorithms to compute the gradients for this network? What test should you apply to recurrent networks to determine if they can be simulated and trained?

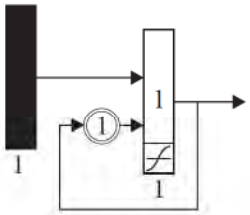


Figure E14.3 Dynamic Network for Exercise E14.7

E14.8 Consider the network in Figure E14.4.

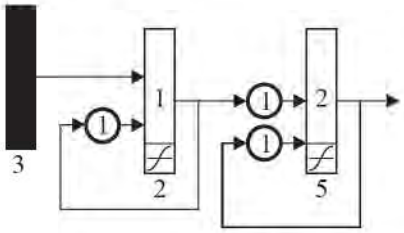


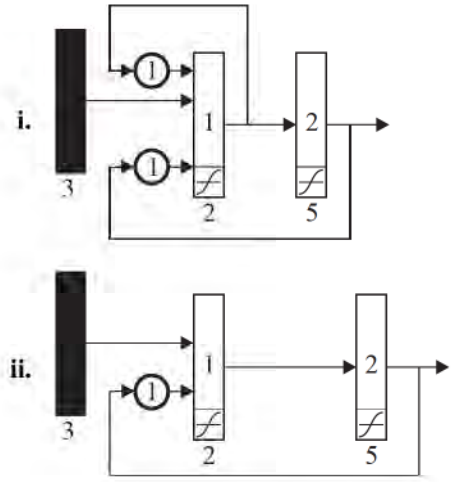
Figure E14.4 Dynamic Network for Exercise E14.8

i. Write out the equations for computing the network response.

ii. Write out the BPTT equations for the network.

iii. Write out the RTRL equations for the network.

E14.9 Repeat E14.8 for the following networks.



E14.10 Consider the network in Figure E14.5.

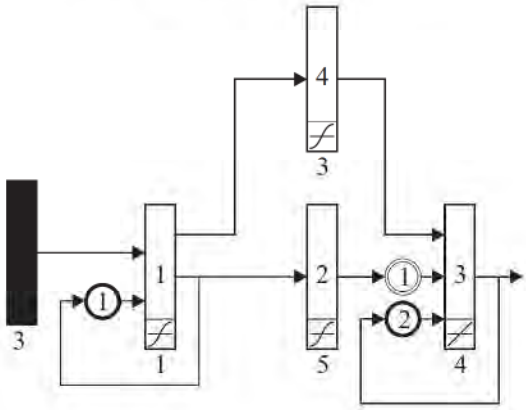


Figure E14.5 Recurrent Network for Exercise E14.10

i. Define the network architecture, by showing U, X, I_m, $DI_{m,1}$, $DL_{m,l}$, L_m^f, L_m^b, $E_{LW}^U(x)$, $E_{LW}^X(u)$.

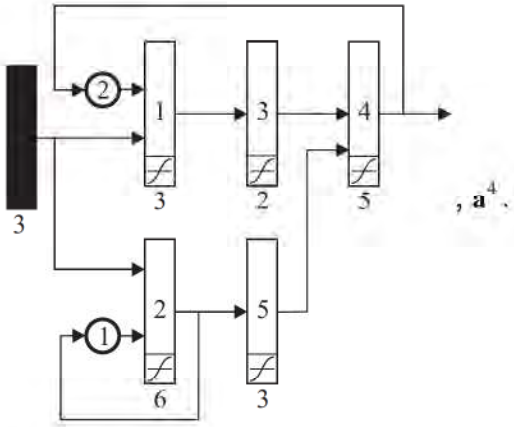
ii. Select a simulation order and write out the equations needed to determine the network response.

iii. Which $S^{u,x}(t)$ will need to be calculated (i.e., for which u and which x)?

iv. Write out Eq. (14.63) specifically for the term $\partial F/\partial \mathbf{a}^3(t)$ and Eq. (14.34) specifically for the term $\partial \mathbf{a}^3(t)/\partial \mathbf{x}^T$. (Expand the summation to show exactly which terms are included.)

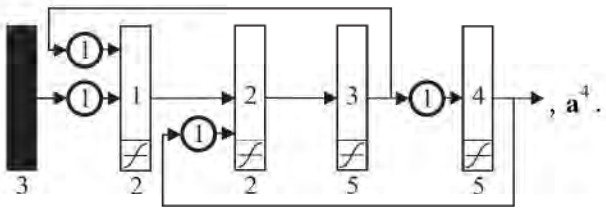
E14.11 Repeat E14.10 for the following networks, except, in part iv., change $\mathbf{a}^3$ to the indicated layer.

i.



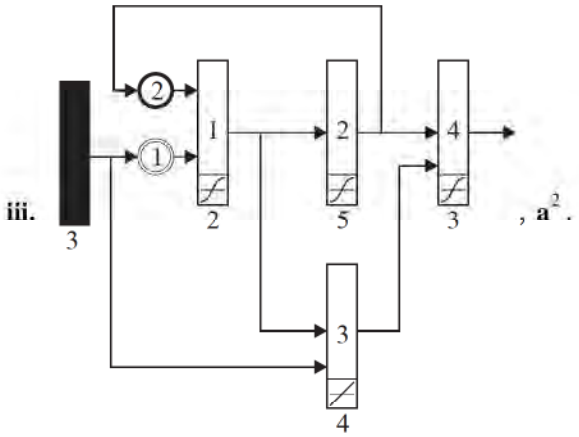
, $\mathbf{a}^4$.

ii.



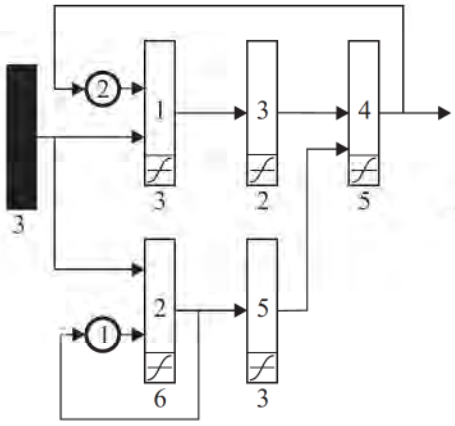
, $\mathbf{a}^4$.

iii.

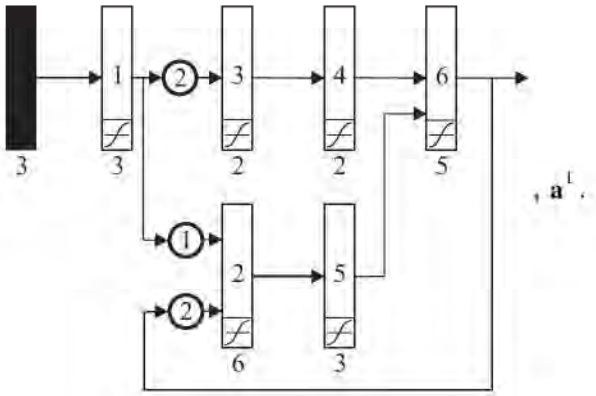


, $\mathbf{a}^2$.

iv.



v.



E14.12 One of the advantages of the RTRL algorithm is that the gradient can be computed at the same time as the network response, since it is computed by starting at the first time point, and then working forward through time. Therefore, RTRL allows a convenient framework for on-line implementation. Suppose that you are implementing the RTRL algorithm, and you update the weights of the network at each time step.

i. Discuss the accuracy of the gradient calculation if the weights are updated at each time step of a steepest descent algorithm.

```
» 2 + 2
ans =
     4
```

ii. Implement the RTRL algorithm in MATLAB for the network in Figure 14.4. Generate training data by using the network with the two weights set equal to 0.5. Use the same input sequence shown in Figure 14.10, and use the network responses $a(t)$ as the targets. Using this data set, train the network using the steepest descent algorithm, with a learning rate of $\alpha = 0.1$. Set the initial weights to zero. Update the weights at each time step. Save the gradient at the end of the eighth time step.

iii. Repeat part ii., but do not update the weights. Compute the gradient at the end of the eighth time step. Compare this gradient with the one you obtained in part ii. Explain any differences

E14.13 Consider again the recurrent network of Figure 14.4. Assume that $F(\mathbf{x})$ is the sum squared error for the first two time points.

i. Let $a(0) = 0$. Find $a(1)$ and $a(2)$ as a functions of $p(1)$, $p(2)$, and the weights in the network.

ii. Find the sum squared error for the first two time steps as an explicit function of $p(1)$, $p(2)$, $t(1)$, $t(2)$, and the network weights.

iii. Using part (ii), find $\dfrac{\partial F}{\partial lw_{1,1}(1)}$.

iv. Compare the results of part (iii) with the results determined by RTRL on page 14-22 and by BPTT on page 14-30.

E14.14 In the process of deriving the RTRL algorithm in Exercise E14.5, you should have produced the following expression

$$\frac{\partial^e \mathbf{a}^3(t)}{\partial vec(\mathbf{LW}^{2,1}(1))^T} = [\mathbf{a}^1(t-1)]^T \otimes \mathbf{S}^{3,2}(t)$$

If

$$\mathbf{a}^1(1) = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } \mathbf{S}^{3,2}(2) = \begin{bmatrix} 2 & 3 \\ 4 & -5 \end{bmatrix},$$

Find $\dfrac{\partial^e \mathbf{a}^3(2)}{\partial vec(\mathbf{LW}^{2,1}(1))^T}$ and indicate $\dfrac{\partial^e a_1^3(2)}{\partial vec(lw_{1,2}^{2,1}(1))^T}$.

E14.15 Each layer of a standard LDDN network has a summing junction, which combines contributions from inputs, other layers, and the bias, as in Eq. (14.1), which is repeated here:

$$\mathbf{n}^m(t) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d) + \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \mathbf{b}^m.$$

If, instead of *summing*, the net input was computed as a *product* of the contributions, how would the RTRL and BPTT algorithms change?

E14.16 As discussed in Exercise E14.15, the contribution to the net input from other layers is computed as product of a layer matrix with a layer output, as in

$$\mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d).$$

If, instead of multiplying the layer matrix times the layer output, we were to compute the distance between each row of the weight matrix and the layer output, as in

$$n_i = -\|_i\mathbf{w} - \mathbf{a}\| .$$

How would the RTRL and BPTT algorithms change?

E14.17 Find and compare the computational complexity for the BPTT and RTRL algorithms applied to the sample network in Figure E14.6 as a function of the number of neurons in Layer 2 (S^2), the number of delays in the tapped delay line (D) and the length of the training sequence (Q).

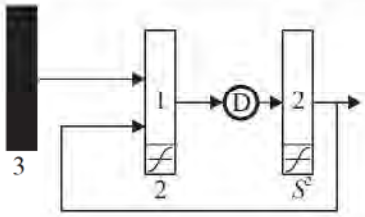


Figure E14.6 Recurrent Network for Exercise E14.17

E14.18 Consider again the network of Figure 14.4. Let the input weight of the network $iw_{1,1} = 1$. Assume that the initial network output is $a(0) = 0$.

i. Write the network output at time t as a function only of the layer weight $lw_{1,1}(1)$, and the input sequence. (The result should be a polynomial in $lw_{1,1}(1)$.)



ii. Find the network output at time $t = 8$, using $lw_{1,1}(1) = -1.4$ and the following input sequence:

$$p(t) = \{3, 1, 1, 6, 3, 5, 1, 6\} .$$

iii. With $lw_{1,1}(1) = -1.4$, the network should be unstable, since this weight in the feedback loop is greater than one in magnitude. The output would generally grow with time for an unstable network. (This applies to linear networks.) In ii., you should have found a small value for $a(8)$. Can you explain this result? (Hint: Investigate the roots of the polynomial you found in part i. You can use the MATLAB command `roots`.) How might this result be related to the spurious valleys in the error surface discussed on page 14-32?

15 Associative Learning

Objectives	15-1
Theory and Examples	15-2
Simple Associative Network	15-3
Unsupervised Hebb Rule	15-5
Hebb Rule with Decay	15-7
Simple Recognition Network	15-9
Instar Rule	15-11
Kohonen Rule	15-15
Simple Recall Network	15-16
Outstar Rule	15-17
Summary of Results	15-21
Solved Problems	15-23
Epilogue	15-34
Further Reading	15-35
Exercises	15-37

Objectives

The neural networks we have discussed so far (in Chapters 4, 7, 10–14) have all been trained in a supervised manner. Each network required a target signal to define correct network behavior.

In contrast, this chapter introduces a collection of simple rules that allow unsupervised learning. These rules give networks the ability to learn associations between patterns that occur together frequently. Once learned, associations allow networks to perform useful tasks such as pattern recognition and recall.

Despite the simplicity of the rules in this chapter, they will form the foundation for powerful networks in Chapters 16, 18, 19.

Theory and Examples

This chapter is all about associations: how associations can be represented by a network, how a network can learn new associations.

What is an association? An association is any link between a system's input and output such that when a pattern A is presented to the system it will respond with pattern B. When two patterns are linked by an association, the input pattern is often referred to as the *stimulus*. Likewise, the output pattern is referred to as the *response*.

Stimulus
Response

Associations are so fundamental that they formed the foundation of the behaviorist school of psychology. This branch of psychology attempted to explain much of animal and human behavior by using associations and rules for learning associations. (This approach has since been largely discredited.)

One of the earliest influences on the behaviorist school of psychology was the classic experiment of Ivan Pavlov, in which he trained a dog to salivate at the sound of a bell, by ringing the bell whenever food was presented. This is an example of what is now called classical conditioning. B. F. Skinner was one of the most influential proponents of the behaviorist school. His classic experiment involved training a rat to press a bar in order to obtain a food pellet. This is an example of instrumental conditioning.

It was to provide a biological explanation for some of this behavior that led Donald Hebb to his postulate, previously quoted in Chapter 7 [Hebb49]:

"*When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*"

In Chapter 7 we analyzed the performance of a supervised learning rule based on Hebb's postulate. In this chapter we will discuss unsupervised forms of Hebbian learning, as well as other related associative learning rules.

A number of researchers have contributed to the development of associative learning. In particular, Tuevo Kohonen, James Anderson and Stephen Grossberg have been very influential. Anderson and Kohonen independently developed the linear associator network in the late 1960s and early 1970s ([Ande72], [Koho72]). Grossberg introduced nonlinear continuous-time associative networks during the same time period (e.g., [Gross68]). All of these researchers, in addition to many others, have continued the development of associative learning up to the present time.

In this chapter we will discuss the elemental associative learning rules. Then, in Chapters 14–16 we will present more complex networks that use

associative learning as a primary component. Chapter 16 will describe Kohonen networks, and Chapters 18 and 19 will discuss Grossberg networks.

Simple Associative Network

Let's take a look at the simplest network capable of implementing an association. An example is the single-input hard limit neuron shown in Figure 15.1.

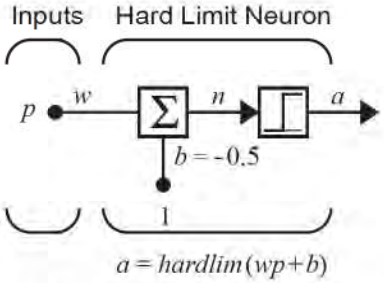


Figure 15.1 Single-Input Hard Limit Associator

The neuron's output a is determined from its input p according to

$$a = hardlim(wp + b) = hardlim(wp - 0.5). \tag{15.1}$$

For simplicity, we will restrict the value of p to be either 0 or 1, indicating whether a stimulus is absent or present. Note that a is limited to the same values by the hard limit transfer function. It indicates the presence or absence of the network's response.

$$p = \begin{cases} 1, & \text{stimulus} \\ 0, & \text{no stimulus} \end{cases} \qquad a = \begin{cases} 1, & \text{response} \\ 0, & \text{no response} \end{cases} \tag{15.2}$$

The presence of an association between the stimulus $p = 1$, and the response $a = 1$ is dictated by the value of w. The network will respond to the stimulus only if w is greater than $-b$ (in this case 0.5).

The learning rules discussed in this chapter are normally used in the framework of a larger network, such as the competitive networks of Chapters 16, 18 and 19. In order to demonstrate the operation of the associative learning rules, without using complex networks, we will use simple networks that have two types of inputs.

Unconditioned Stimulus

Conditioned Stimulus

One set of inputs will represent the *unconditioned stimulus*. This is analogous to the food presented to the dog in Pavlov's experiment. Another set of inputs will represent the *conditioned stimulus*. This is analogous to the bell in Pavlov's experiment. Initially the dog salivates only when food is

presented. This is an innate characteristic that does not have to be learned. However, when the bell is repeatedly paired with the food, the dog is conditioned to salivate at the sound of the bell, even when no food is present.

We will represent the unconditioned stimulus as $\mathbf{p}^0$ and the conditioned stimulus simply as $\mathbf{p}$. For our purposes we will assume that the weights associated with $\mathbf{p}^0$ are fixed, but that the weights associated with $\mathbf{p}$ are adjusted according to the relevant learning rule.



Figure 15.2 shows a network for recognizing bananas. The network has both an unconditioned stimulus (banana shape) and a conditioned stimulus (banana smell). We don't mean to imply here that smell is more conditionable than sight. In our examples in this chapter the choices of conditioned and unconditioned stimuli are arbitrary and are used simply to demonstrate the performance of the learning rules. We will use this network to demonstrate the operation of the Hebb rule in the following section.

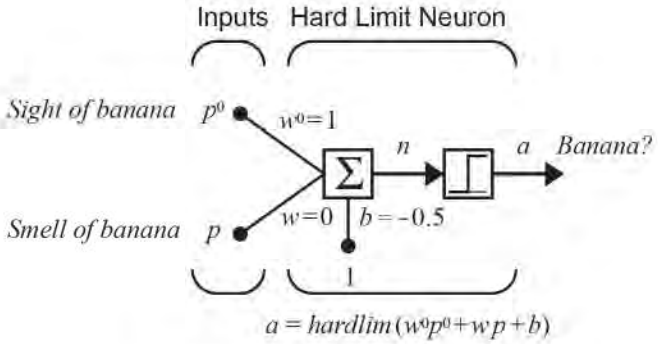
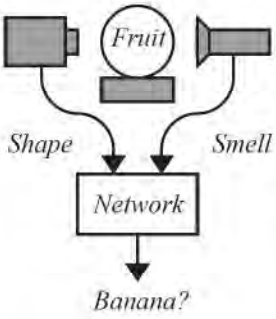


Figure 15.2 Banana Associator



The definitions of the unconditioned and conditioned inputs for this network are

$$p^0 = \begin{cases} 1, & \text{shape detected} \\ 0, & \text{shape not detected} \end{cases} \qquad p = \begin{cases} 1, & \text{smell detected} \\ 0, & \text{smell not detected} \end{cases} . \tag{15.3}$$

At this time we would like the network to associate the shape of a banana, but the not the smell, with a response indicating the fruit is a banana. The problem is solved by assigning a value greater than $-b$ to w^0 and assigning a value less than $-b$ to w. The following values satisfy these requirements:

$$w^0 = 1, \; w = 0. \tag{15.4}$$

The banana associator's input/output function now simplifies to

$$a = hardlim(p^0 - 0.5). \tag{15.5}$$

Thus, the network will only respond if a banana is sighted ($p^0 = 1$), whether a banana is smelled ($p = 1$) or not ($p = 0$).

We will use this network in later sections to illustrate the performance of several associative learning rules.

Unsupervised Hebb Rule

For simple problems it is not difficult to design a network with a fixed set of associations. On the other hand, a more useful network would be able to learn associations.

When should an association be learned? It is generally accepted that both animals and humans tend to associate things that occur simultaneously. To paraphrase Hebb: if a banana smell stimulus occurs simultaneously with a banana concept response (activated by some other stimulus such as the sight of a banana shape), the network should strengthen the connection between them so that later it can activate its banana concept in response to the banana smell alone.

The unsupervised Hebb rule does just that by increasing the weight w_{ij} between a neuron's input p_j and output a_i in proportion to their product:

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q) p_j(q). \tag{15.6}$$

(See also Eq. (7.5).) The learning rate α dictates how many times a stimulus and response must occur together before an association is made. In the network in Figure 15.2, an association will be made when $w > -b = 0.5$, since then $p = 1$ will produce the response $a = 1$, regardless of the value of p^0.

Local Learning

Note that Eq. (15.6) uses only signals available within the layer containing the weights being updated. Rules that satisfy this condition are called *local learning* rules. This is in contrast to the backpropagation rule, for example, in which the sensitivity must be propagated back from the final layer. The rules introduced in this chapter will all be local learning rules.

The unsupervised Hebb rule can also be written in vector form:

$$\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha \mathbf{a}(q)\mathbf{p}^T(q). \tag{15.7}$$

Training Sequence

As with all unsupervised rules, learning is performed in response to a series of inputs presented in time (the *training sequence*):

$$\mathbf{p}(1), \mathbf{p}(2), \ldots, \mathbf{p}(Q). \tag{15.8}$$

(Note that we are using the notation $\mathbf{p}(q)$, instead of $\mathbf{p}_q$, in order to emphasize the time-sequence nature of the inputs.) At each iteration, the output $\mathbf{a}$ is calculated in response to the input $\mathbf{p}$, and then the weights $\mathbf{W}$ are updated with the Hebb rule.



Let's apply the unsupervised Hebb rule to the banana associator. The associator will start with the weight values determined in our previous example, so that it will initially respond to the sight, but not the smell, of a banana.

$$w^0 = 1, w(0) = 0 \tag{15.9}$$

The associator will be repeatedly exposed to a banana. However, while the network's smell sensor will work reliably, the shape sensor will operate only intermittently (on even time steps). Thus the training sequence will consist of repetitions of the following two sets of inputs:

$$\{p^0(1) = 0, p(1) = 1\}, \{p^0(2) = 1, p(2) = 1\}, \ldots . \tag{15.10}$$

The first weight w^0, representing the weight for the unconditioned stimulus p^0, will remain constant, while w will be updated at each iteration, using the unsupervised Hebb rule with a learning rate of 1:

$$w(q) = w(q-1) + a(q)p(q) . \tag{15.11}$$

The output for the first iteration ($q = 1$) is

$$\begin{aligned} a(1) &= hardlim(w^0 p^0(1) + w(0)p(1) - 0.5) \\ &= hardlim(1 \cdot 0 + 0 \cdot 1 - 0.5) = 0 \quad \text{(no response)} . \end{aligned} \tag{15.12}$$

The smell alone did not generate a response. Without a response, the Hebb rule does not alter w.

$$w(1) = w(0) + a(1)p(1) = 0 + 0 \cdot 1 = 0 \tag{15.13}$$

In the second iteration, both the banana's shape and smell are detected and the network responds accordingly:

$$\begin{aligned} a(2) &= hardlim(w^0 p^0(2) + w(1)p(2) - 0.5) \\ &= hardlim(1 \cdot 1 + 0 \cdot 1 - 0.5) = 1 \quad \text{(banana)} . \end{aligned} \tag{15.14}$$

Because the smell stimulus and the response have occurred simultaneously, the Hebb rule increases the weight between them.

$$w(2) = w(1) + a(2)p(2) = 0 + 1 \cdot 1 = 1 \tag{15.15}$$

When the sight detector fails again, in the third iteration, the network responds anyway. It has made a useful association between the smell of a banana and its response.

$$a(3) = hardlim(w^0 p^0(3) + w(2)p(3) - 0.5) \qquad (15.16)$$
$$= hardlim(1 \cdot 0 + 1 \cdot 1 - 0.5) = 1 \quad \text{(banana)}$$

$$w(3) = w(2) + a(3)p(3) = 1 + 1 \cdot 1 = 2 \qquad (15.17)$$

From now on, the network is capable of responding to bananas that are detected either by sight or smell. Even if both detection systems suffer intermittent faults, the network will be correct most of the time.



To experiment with the unsupervised Hebb rule, use the Neural Network Design Demonstration Unsupervised Hebb Rule (**nnd13uh**).

We have seen that the unsupervised Hebb rule can learn useful associations. However, the Hebb rule, as defined in Eq. (15.6), has some practical shortcomings. The first problem becomes evident if we continue to present inputs and update w in the example above. The weight w will become arbitrarily large. This is at odds with the biological systems that inspired the Hebb rule. Synapses cannot grow without bound.

The second problem is that there is no mechanism for weights to decrease. If the inputs or outputs of a Hebb network experience any noise, every weight will grow (however slowly) until the network responds to any stimulus.

Hebb Rule with Decay

One way to improve the Hebb rule is by adding a weight decay term (Eq. (7.45)),

$$\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha \mathbf{a}(q)\mathbf{p}^T(q) - \gamma \mathbf{W}(q-1)$$
$$= (1-\gamma)\mathbf{W}(q-1) + \alpha \mathbf{a}(q)\mathbf{p}^T(q)\,, \qquad (15.18)$$

Decay Rate

where γ, the *decay rate*, is a positive constant less than one. As γ approaches zero, the learning law becomes the standard rule. As γ approaches one, the learning law quickly forgets old inputs and remembers only the most recent patterns. This keeps the weight matrix from growing without bound. (The idea of filtering the weight changes was also discussed in Chapter 12, where we called it momentum.)

The maximum weight value w_{ij}^{max} is determined by γ. This value is found by setting both a_i and p_j to a value of 1 for all q (to maximize learning) in the scalar version of Eq. (15.18) and solving for the steady state weight (i.e. when both new and old weights are equal).

$$w_{ij} = (1-\gamma)w_{ij} + \alpha a_i p_j$$
$$w_{ij} = (1-\gamma)w_{ij} + \alpha \qquad (15.19)$$
$$w_{ij} = \frac{\alpha}{\gamma}$$



Let's examine the operation of the Hebb rule with decay on our previous banana associator problem. We will use a decay rate γ of 0.1. The first iteration, where only the smell stimulus is presented, is the same:

$$a(1) = 0 \quad \text{(no response)}, \quad w(1) = 0. \qquad (15.20)$$

The next iteration also produces identical results. Here both stimuli are presented, and the network responds to the shape. Coincidence of the smell stimulus and response create a new association:

$$a(2) = 1 \quad \text{(banana)}, \quad w(2) = 1. \qquad (15.21)$$

The results of the third iteration are not the same. The network has learned to respond to the smell, and the weight continues to increase. However, this time the weight increases by only 0.9, instead of 1.0.

$$w(3) = w(2) + a(3)p(3) - 0.1w(2) = 1 + 1 \cdot 1 - 0.1 \cdot 1 = 1.9 \qquad (15.22)$$

The decay term limits the weight's value, so that no matter how often the association is reinforced, w will never increase beyond w_{ij}^{max}.

$$w_{ij}^{max} = \frac{\alpha}{\gamma} = \frac{1}{0.1} = 10 \qquad (15.23)$$

The new rule also ensures that associations learned by the network will not be artifacts of noise. Any small random increases will soon decay away.

Figure 15.3 displays the response of the Hebb rule, with and without decay, for the banana recognition example. Without decay, the weight continues to increase by the same amount each time the neuron is activated. When decay is added, the weight exponentially approaches its maximum value ($w_{ij}^{max} = 10$).



To experiment with the Hebb rule with decay, use the Neural Network Design Demonstrations Hebb with Decay(**nnd13hd**) *and Effect of Decay Rate* (**nnd13edr**).

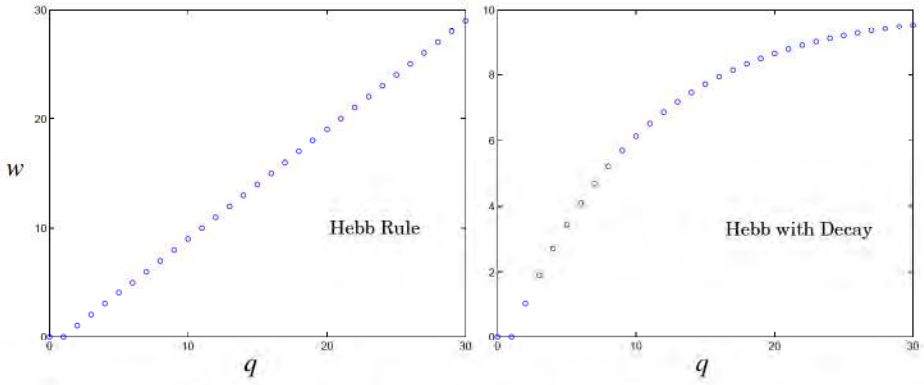


Figure 15.3 Response of the Hebb Rule, With and Without Decay

The Hebb rule with decay does solve the problem of large weights. However, it does so at a price. The environment must be counted on to occasionally present all stimuli that have associations. Without reinforcement, associations will decay away.

To illustrate this fact, consider Eq. (15.18) if $a_i = 0$:

$$w_{ij}(q) = (1-\gamma)w_{ij}(q-1). \tag{15.24}$$

If $\gamma = 0.1$, this reduces to

$$w_{ij}(q) = (0.9)w_{ij}(q-1). \tag{15.25}$$

Therefore w_{ij} will be decreased by 10% at each presentation for which $a_i = 0$. Any association that was previously learned will eventually be lost. We will discuss a solution to this problem in a later section.

Simple Recognition Network

Instar

So far we have considered only associations between scalar inputs and outputs. We will now examine a neuron that has a vector input. (See Figure 15.4.) This neuron, which is sometimes referred to as an *instar*, is the simplest network that is capable of pattern recognition, as we will demonstrate shortly.

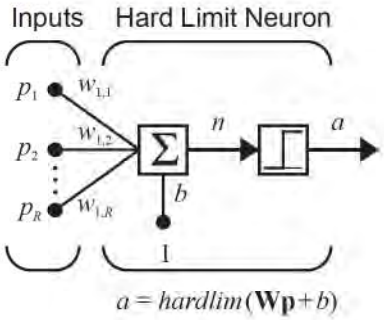


Figure 15.4 Instar

You will notice the similarities between the instar of Figure 15.4 and the perceptron of Figure 4.2 (also the ADALINE of Figure 10.2 and the linear associator of Figure 7.1). We give these networks different names, in part for historical reasons (since they arose at different times and out of different environments), and because they perform different functions and are analyzed in different ways. For example, we will not directly consider the decision boundary of the instar, although this was an important concept for the perceptron. Instead, we will analyze the ability of the instar to recognize a pattern, as with the neurons in the first layer of the Hamming network. (See page 3-10.)

The input/output expression for the instar is

$$a = hardlim(\mathbf{W}\mathbf{p} + b) = hardlim({}_1\mathbf{w}^T\mathbf{p} + b). \tag{15.26}$$

The instar will be active whenever the inner product between the weight vector (row of the weight matrix) and the input is greater than or equal to $-b$:

$${}_1\mathbf{w}^T\mathbf{p} \geq -b. \tag{15.27}$$

From our discussion of the Hamming network on page 3-10, we know that for two vectors of constant length, the inner product will be largest when they point in the same direction. We can also show this using Eq. (5.15):

$${}_1\mathbf{w}^T\mathbf{p} = \|{}_1\mathbf{w}\|\|\mathbf{p}\|\cos\theta \geq -b, \tag{15.28}$$

where θ is the angle between the two vectors. Clearly the inner product is maximized when the angle θ is 0. If $\mathbf{p}$ and ${}_1\mathbf{w}$ have the same length ($\|\mathbf{p}\| = \|{}_1\mathbf{w}\|$), then the inner product will be largest when $\mathbf{p} = {}_1\mathbf{w}$.

Based on these arguments, the instar of Figure 15.4 will be active when $\mathbf{p}$ is "close" to ${}_1\mathbf{w}$. By setting the bias b appropriately, we can select how close the input vector must be to the weight vector in order to activate the instar.

If we set

$$b = -\|_1\mathbf{w}\|\|\mathbf{p}\|, \tag{15.29}$$

then the instar will only be active when $\mathbf{p}$ points in exactly the same direction as ${}_1\mathbf{w}$ ($\theta = 0$). Thus, we will have a neuron that recognizes only the pattern ${}_1\mathbf{w}$.

If we would like the instar to respond to any pattern near ${}_1\mathbf{w}$ (θ small), then we can increase b to some value larger than $-\|_1\mathbf{w}\|\|\mathbf{p}\|$. The larger the value of b, the more patterns there will be that can activate the instar, thus making it the less discriminatory.

We should note that this analysis assumes that all input vectors have the same length (norm). We will revisit the question of normalization in Chapters 16, 18 and 19.

We can now design a vector recognition network if we know which vector we want to recognize. However, if the network is to learn a vector without supervision, we need a new rule, since neither version of the Hebb rule produces normalized weights.

Instar Rule

One problem of the Hebb rule with decay was that it required stimuli to be repeated or associations would be lost. A better rule might allow weight decay only when the instar is active ($a \neq 0$). Weight values would still be limited, but forgetting would be minimized. Consider again the original Hebb rule:

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q). \tag{15.30}$$

To get the benefits of weight decay, while limiting the forgetting problem, a decay term can be added that is proportional to $a_i(q)$:

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q) - \gamma a_i(q)w_{ij}(q-1) \tag{15.31}$$

We can simplify Eq. (15.31) by setting γ equal to α (so new weight values are learned at the same rate old values decay) and gathering terms.

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)(p_j(q) - w_{ij}(q-1)) \tag{15.32}$$

Instar Rule

This equation, called the *instar rule*, can also be rewritten in vector form:

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha a_i(q)(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)). \tag{15.33}$$

The performance of the instar rule can be best understood if we consider the case where the instar is active ($a_i = 1$). Eq. (15.33) can then be written

$$
\begin{aligned}
{}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\
&= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)\,.
\end{aligned} \tag{15.34}
$$

This operation is displayed graphically in Figure 15.5.

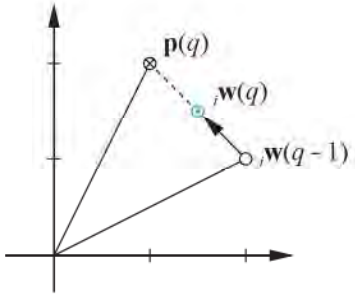


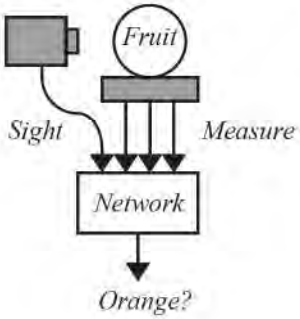
Figure 15.5 Graphical Representation of the Instar Rule

When the instar is active, the weight vector is moved toward the input vector along a line between the old weight vector and the input vector. The distance the weight vector moves depends on the value of α. When $\alpha = 0$, the new weight vector is equal to the old weight vector (no movement). When $\alpha = 1$, the new weight vector is equal to the input vector (maximum movement). If $\alpha = 0.5$, the new weight vector will be halfway between the old weight vector and the input vector.

One useful feature of the instar rule is that if the input vectors are normalized, then ${}_i\mathbf{w}$ will also be normalized once it has learned a particular vector $\mathbf{p}$. We have found a rule that not only minimizes forgetting, but results in normalized weight vectors, if the input vectors are normalized.



Let's apply the instar rule to the network in Figure 15.6. It has two inputs: one indicating whether a fruit has been visually identified as an orange (unconditioned stimulus) and another consisting of the three measurements taken of the fruit (conditioned stimulus).



The output of this network is

$$
a = hardlim(w^0 p^0 + \mathbf{W}\mathbf{p} + b)\,. \tag{15.35}
$$

The elements of input $\mathbf{p}$ will be constrained to ± 1 values, as defined in Chapter 3 (Eq. (3.2)). This constraint ensures that $\mathbf{p}$ is a normalized vector with a length of $\|\mathbf{p}\| = \sqrt{3}$. The definitions of p^0 and $\mathbf{p}$ are

$$
p^0 = \begin{cases} 1, & \text{orange detected visually} \\ 0, & \text{orange not detected} \end{cases} \qquad \mathbf{p} = \begin{bmatrix} shape \\ texture \\ weight \end{bmatrix}. \tag{15.36}
$$

The bias b is -2, a value slightly more positive than $-\|\mathbf{p}\|^2 = -3$. (See Eq. (15.29).)

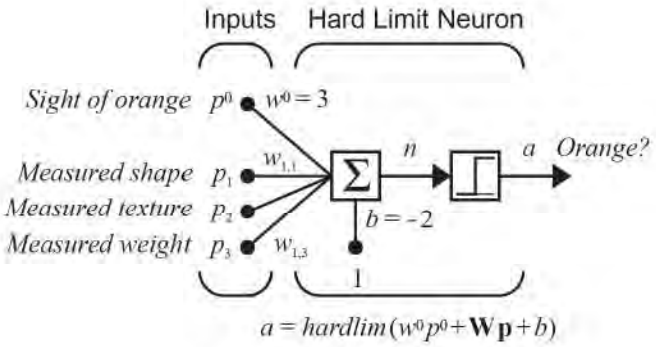


Figure 15.6 Orange Recognizer

We would like the network to have a constant association between the sight of an orange and its response, so w^0 will be set greater than $-b$. But initially, the network should not respond to any combination of fruit measurements, so the measurement weights will start with values of 0.

$$w^0 = 3, \quad \mathbf{W}(0) = {}_1\mathbf{w}^T(0) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \tag{15.37}$$

The measurement weights will be updated with the instar rule, using a learning rate of $\alpha = 1$.

$${}_1\mathbf{w}(q) = {}_1\mathbf{w}(q-1) + a(q)(\mathbf{p}(q) - {}_1\mathbf{w}(q-1)) \tag{15.38}$$

The training sequence will consist of repeated presentations of an orange. The measurements will be given every time. However, in order to demonstrate the operation of the instar rule, we will assume that the visual system only operates correctly on even time steps, due to a fault in its construction.

$$\left\{ p^0(1) = 0, \mathbf{p}(1) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \left\{ p^0(2) = 1, \mathbf{p}(2) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \ldots \tag{15.39}$$

Because $\mathbf{W}$ initially contains all zeros, the instar does not respond to the measurements of an orange in the first iteration.

$$a(1) = hardlim(w^0 p^0(1) + \mathbf{W}\mathbf{p}(1) - 2)$$

$$a(1) = hardlim\left(3 \cdot 0 + \begin{bmatrix}0 & 0 & 0\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} - 2\right) = 0 \quad \text{(no response)} \tag{15.40}$$

Since the neuron did not respond, its weights ${}_1\mathbf{w}$ are not altered by the instar rule.

$${}_1\mathbf{w}(1) = {}_1\mathbf{w}(0) + a(1)(\mathbf{p}(1) - {}_1\mathbf{w}(0)) \tag{15.41}$$

$$= \begin{bmatrix}0\\0\\0\end{bmatrix} + 0\left(\begin{bmatrix}1\\-1\\-1\end{bmatrix} - \begin{bmatrix}0\\0\\0\end{bmatrix}\right) = \begin{bmatrix}0\\0\\0\end{bmatrix}$$

However, the neuron does respond when the orange is identified visually, in addition to being measured, in the second iteration.

$$a(2) = hardlim(w^0 p^0(2) + \mathbf{W}\mathbf{p}(2) - 2) \tag{15.42}$$

$$= hardlim\left(3 \cdot 1 + \begin{bmatrix}0 & 0 & 0\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} - 2\right) = 1 \quad \text{(orange)}$$

The result is that the neuron learns to associate the orange's measurement vector with its response. The weight vector ${}_1\mathbf{w}$ becomes a copy of the orange measurement vector.

$${}_1\mathbf{w}(2) = {}_1\mathbf{w}(1) + a(2)(\mathbf{p}(2) - {}_1\mathbf{w}(1)) \tag{15.43}$$

$$= \begin{bmatrix}0\\0\\0\end{bmatrix} + 1\left(\begin{bmatrix}1\\-1\\-1\end{bmatrix} - \begin{bmatrix}0\\0\\0\end{bmatrix}\right) = \begin{bmatrix}1\\-1\\-1\end{bmatrix}$$

The network can now recognize the orange by its measurements. The neuron responds in the third iteration, even though the visual detection system failed again.

$$a(3) = hardlim(w^0 p^0(3) + \mathbf{W}\mathbf{p}(3) - 2)$$

$$a(3) = hardlim\left(3 \cdot 0 + \begin{bmatrix}1 & -1 & -1\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} - 2\right) = 1 \quad \text{(orange)} \tag{15.44}$$

Having completely learned the measurements, the weights stop changing. (A lower learning rate would have required more iterations.)

$${}_1\mathbf{w}(3) = {}_1\mathbf{w}(2) + a(3)(\mathbf{p}(3) - {}_1\mathbf{w}(2)) \tag{15.45}$$

$$= \begin{bmatrix}1\\-1\\-1\end{bmatrix} + 1\left(\begin{bmatrix}1\\-1\\-1\end{bmatrix} - \begin{bmatrix}1\\-1\\-1\end{bmatrix}\right) = \begin{bmatrix}1\\-1\\-1\end{bmatrix}$$

The network has learned to recognize an orange by its measurements, even when its visual detection system fails.



To experiment with the instar rule, use the Neural Network Design Demonstrations Instar (`nnd13is`) *and Graphical Instar* (`nnd13gis`).

Kohonen Rule

Kohonen Rule

At this point it is appropriate to introduce another associative learning rule, which is related to the instar rule. It is the *Kohonen rule*:

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)), \quad \text{for } i \in X(q). \tag{15.46}$$

Like the instar rule, the Kohonen rule allows the weights of a neuron to learn an input vector and is therefore suitable for recognition applications. Unlike the instar rule, learning is not proportional to the neuron's output $a_i(q)$. Instead, learning occurs when the neuron's index i is a member of the set $X(q)$.

If the instar rule is applied to a layer of neurons whose transfer function only returns values of 0 or 1 (such as *hardlim*), then the Kohonen rule can be made equivalent to the instar rule by defining $X(q)$ as the set of all i such that $a_i(q) = 1$. The advantage of the Kohonen rule is that it can also be used with other definitions. It is useful for training networks such as the self-organizing feature map, which will be introduced in Chapter 16.

Simple Recall Network

Outstar

We have seen that the instar network (with a vector input and a scalar output) can perform pattern recognition by associating a particular vector stimulus with a response. The *outstar* network, shown in Figure 15.7, has a scalar input and a vector output. It can perform *pattern recall* by associating a stimulus with a vector response.

The input-output expression for this network is

$$\mathbf{a} = \mathbf{satlins}(\mathbf{W}p). \tag{15.47}$$

The symmetric saturating function **satlins** was chosen because this network will be used to recall a vector containing values of -1 or 1.

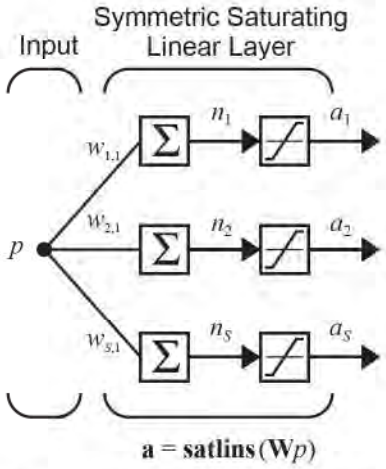


Figure 15.7 Outstar Network

If we would like the network to associate a stimulus (an input of 1) with a particular output vector $\mathbf{a}^*$, we can simply set $\mathbf{W}$ (which contains only a single column vector) equal to $\mathbf{a}^*$. Then, if p is 1, the output will be $\mathbf{a}^*$:

$$\mathbf{a} = \mathbf{satlins}(\mathbf{W}p) = \mathbf{satlins}(\mathbf{a}^* \cdot 1) = \mathbf{a}^*. \tag{15.48}$$

(This assumes that the elements of $\mathbf{a}^*$ are less than or equal to 1 in magnitude.)

Note that we have created a recall network by setting a *column* of the weight matrix to the desired vector. Earlier we designed a recognition network by setting a *row* of the weight matrix to the desired vector.

We can now design a network that can recall a known vector $\mathbf{a}^*$, but we need a learning rule if the network is to learn a vector without supervision. We will describe such a learning rule in the next section.

Outstar Rule

To derive the instar rule, forgetting was limited by making the weight decay term of the Hebb rule proportional to the output of the network, a_i. Conversely, to obtain the outstar learning rule, we make the weight decay term proportional to the input of the network, p_j:

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q)p_j(q) - \gamma p_j(q)w_{ij}(q-1). \tag{15.49}$$

If we set the decay rate γ equal to the learning rate α and collect terms, we get

$$w_{ij}(q) = w_{ij}(q-1) + \alpha(a_i(q) - w_{ij}(q-1))p_j(q). \tag{15.50}$$

The outstar rule has properties complimentary to the instar rule. Learning occurs whenever p_j is nonzero (instead of a_i). When learning occurs, column $\mathbf{w}_j$ moves toward the output vector.

Outstar Rule

As with the instar rule, the *outstar rule* can be written in vector form:

$$\mathbf{w}_j(q) = \mathbf{w}_j(q-1) + \alpha(\mathbf{a}(q) - \mathbf{w}_j(q-1))p_j(q), \tag{15.51}$$

where $\mathbf{w}_j$ is the jth column of the matrix $\mathbf{W}$.

To test the outstar rule we will train the network shown in Figure 15.8.

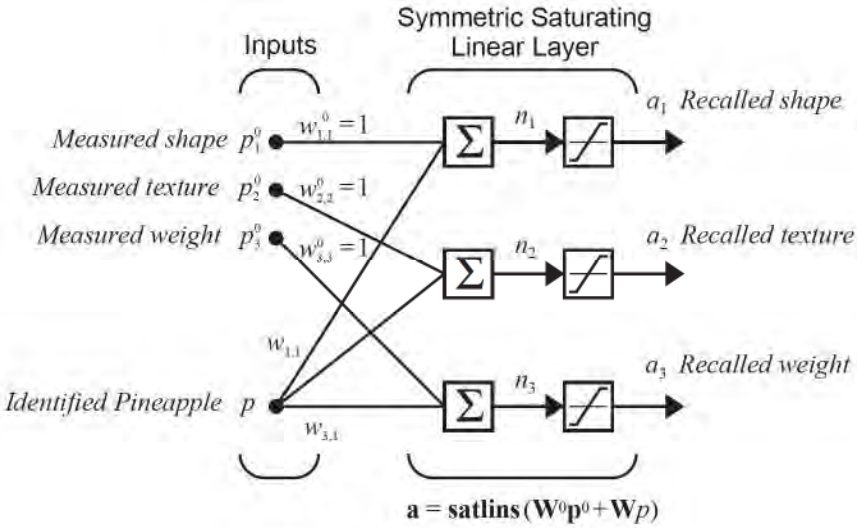
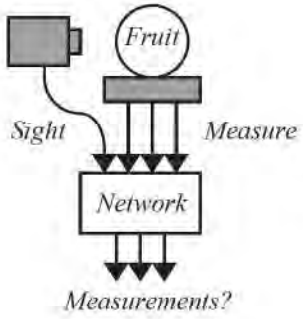


Figure 15.8 Pineapple Recaller

The outputs of the network are calculated as follows:

$$\mathbf{a} = \mathbf{satlins}(\mathbf{W}^0\mathbf{p}^0 + \mathbf{W}p), \tag{15.52}$$

where



$$\mathbf{W}^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{15.53}$$

The network's two inputs provide it with measurements $\mathbf{p}^0$ taken on a fruit (unconditioned stimulus), as well as a signal p indicating a pineapple has been identified visually (conditioned stimulus).

$$\mathbf{p}^0 = \begin{bmatrix} shape \\ texture \\ weight \end{bmatrix} \quad p = \begin{cases} 1, & \text{if a pineapple can be seen} \\ 0, & \text{otherwise} \end{cases} \tag{15.54}$$

The network's output is to reflect the measurements of the fruit currently being examined, using whatever inputs are available.

The weight matrix for the unconditioned stimulus, $\mathbf{W}^0$, is set to the identity matrix, so that any set of measurements $\mathbf{p}^0$ (with ±1 values) will be copied to the output $\mathbf{a}$. The weight matrix for the conditioned stimulus, $\mathbf{W}$, is set to zero initially, so that a 1 on p will not generate a response. $\mathbf{W}$ will be updated with the outstar rule using a learning rate of 1:

$$\mathbf{w}_j(q) = \mathbf{w}_j(q-1) + (\mathbf{a}(q) - \mathbf{w}_j(q-1))p(q). \tag{15.55}$$

The training sequence will consist of repeated presentations of the sight and measurements of a pineapple. The pineapple measurements are

$$\mathbf{p}^{pineapple} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}. \tag{15.56}$$

However, due to a fault in the measuring system, measured values will only be available on even iterations.

$$\left\{\mathbf{p}^0(1) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, p(1) = 1\right\}, \left\{\mathbf{p}^0(2) = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}, p(2) = 1\right\}, \ldots \tag{15.57}$$

In the first iteration, the pineapple is seen, but the measurements are unavailable.

$$\mathbf{a}(1) = \mathbf{satlins}(\mathbf{W}^0\mathbf{p}^0(1) + \mathbf{W}p(1)), \tag{15.58}$$

$$\mathbf{a}(1) = \mathbf{satlins}\left(\begin{bmatrix}0\\0\\0\end{bmatrix} + \begin{bmatrix}0\\0\\0\end{bmatrix}1\right) = \begin{bmatrix}0\\0\\0\end{bmatrix} \quad \text{(no response)} \tag{15.59}$$

The network sees the pineapple but cannot output proper measurements, because it has not learned them and the measurement system is not working. The weights remain unchanged after being updated.

$$\mathbf{w}_1(1) = \mathbf{w}_1(0) + (\mathbf{a}(1) - \mathbf{w}_1(0))p(1) = \begin{bmatrix}0\\0\\0\end{bmatrix} + \left(\begin{bmatrix}0\\0\\0\end{bmatrix} - \begin{bmatrix}0\\0\\0\end{bmatrix}\right)1 = \begin{bmatrix}0\\0\\0\end{bmatrix} \tag{15.60}$$

In the second iteration the pineapple is seen, and the measurements are taken properly.

$$\mathbf{a}(2) = \mathbf{satlins}\left(\begin{bmatrix}-1\\-1\\1\end{bmatrix} + \begin{bmatrix}0\\0\\0\end{bmatrix}1\right) = \begin{bmatrix}-1\\-1\\1\end{bmatrix} \quad \text{(measurements given)} \tag{15.61}$$

The measurements are available, so the network outputs them correctly. The weights are then updated as follows:

$$\begin{aligned}\mathbf{w}_1(2) &= \mathbf{w}_1(1) + (\mathbf{a}(2) - \mathbf{w}_1(1))p(2)\\ &= \begin{bmatrix}0\\0\\0\end{bmatrix} + \left(\begin{bmatrix}-1\\-1\\1\end{bmatrix} - \begin{bmatrix}0\\0\\0\end{bmatrix}\right)1 = \begin{bmatrix}-1\\-1\\1\end{bmatrix}.\end{aligned} \tag{15.62}$$

Since the sight of the pineapple and the measurements were both available, the network forms an association between them. The weight matrix is now a copy of the measurements, so they can be recalled later.

In iteration three, measurements are unavailable once again, but the output is

$$\mathbf{a}(3) = \mathbf{satlins}\left(\begin{bmatrix}0\\0\\0\end{bmatrix} + \begin{bmatrix}-1\\-1\\1\end{bmatrix} 1\right) = \begin{bmatrix}-1\\-1\\1\end{bmatrix} \quad \text{(measurements recalled)}. \quad (15.63)$$

The network is now able to recall the measurements of the pineapple when it sees it, even though the measurement system fails. From now on, the weights will no longer change values unless a pineapple is seen with different measurements.

$$\mathbf{w}_1(3) = \mathbf{w}_1(2) + (\mathbf{a}(2) - \mathbf{w}_1(2))p(2)$$

$$= \begin{bmatrix}-1\\-1\\1\end{bmatrix} + \left(\begin{bmatrix}-1\\-1\\1\end{bmatrix} - \begin{bmatrix}-1\\-1\\1\end{bmatrix}\right) 1 = \begin{bmatrix}-1\\-1\\1\end{bmatrix} \quad (15.64)$$



*To experiment with the outstar rule with decay, use the Neural Network Design Demonstration Outstar Rule (*nnd13os*).*

In Chapter 19 we will investigate the ART networks, which use both the instar and the outstar rules.

Summary of Results

Association

An association is a link between the inputs and outputs of a network so that when a stimulus A is presented to the network, it will output a response B.

Associative Learning Rules

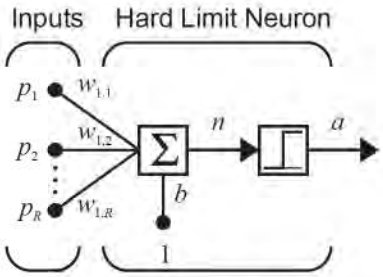
Unsupervised Hebb Rule

$$\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha \mathbf{a}(q)\mathbf{p}^T(q)$$

Hebb Rule with Decay

$$\mathbf{W}(q) = (1-\gamma)\mathbf{W}(q-1) + \alpha \mathbf{a}(q)\mathbf{p}^T(q)$$

Instar



$$a = hardlim(\mathbf{W}\mathbf{p}+b)$$

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b)$$

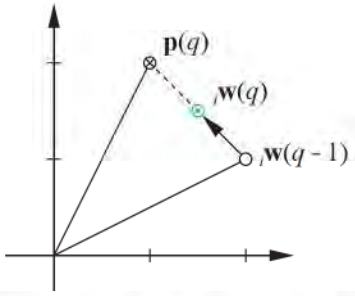
The instar is activated for ${}_1\mathbf{w}^T\mathbf{p} = \|{}_1\mathbf{w}\|\|\mathbf{p}\|\cos\theta \geq -b$,

where θ is the angle between $\mathbf{p}$ and ${}_1\mathbf{w}$.

Instar Rule

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha a_i(q)(\mathbf{p}(q) - {}_i\mathbf{w}(q-1))$$

$${}_i\mathbf{w}(q) = (1-\alpha)\mathbf{w}(q-1) + \alpha \mathbf{p}(q), \quad if(a_i(q) = 1)$$

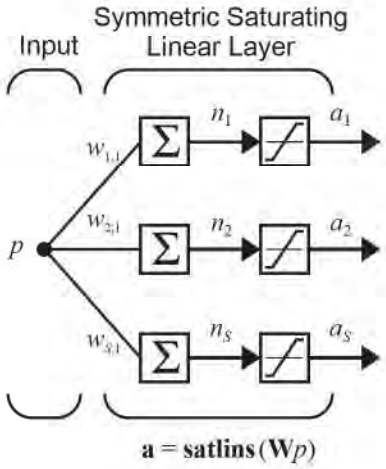


Graphical Representation of the Instar Rule ($a_i(q) = 1$)

Kohonen Rule

$$_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)), \quad \text{for } i \in X(q)$$

Outstar



Outstar Rule

$$\mathbf{w}_j(q) = \mathbf{w}_j(q-1) + \alpha(\mathbf{a}(q) - \mathbf{w}_j(q-1))p_j(q)$$

Solved Problems

P15.1 In Eq. (15.19) the maximum weight for the Hebb rule with decay was calculated, assuming that p_j and a_i were 1 at every time step. Calculate the maximum weight resulting if p_j and a_i alternate together between values of 0 and 1.

We begin with the scalar version of the Hebb rule with decay:

$$w_{ij}(q) = (1-\gamma)w_{ij}(q-1) + \alpha a_i(q)p_j(q).$$

We can rewrite this expression twice using q to index the weight values as the weight is updated over two time steps.

$$w_{ij}(q+1) = (1-\gamma)w_{ij}(q) + \alpha a_i(q)p_j(q)$$

$$w_{ij}(q+2) = (1-\gamma)w_{ij}(q+1) + \alpha a_i(q+1)p_j(q+1)$$

By substituting the first equation into the second, we get a single expression showing how w_{ij} is updated over two time steps.

$$w_{ij}(q+2) = (1-\gamma)((1-\gamma)w_{ij}(q) + \alpha a_i(q)p_j(q)) + \alpha a_i(q+1)p_j(q+1)$$

At this point we can substitute values for p_j and a_i. Because we are looking for a maximum weight, we will set $p_j(q)$ and $a_i(q)$ to 0, and $p_j(q+1)$ and $a_i(q+1)$ to 1. This will mean that the weight decreases in the first time step, and increases in the second, ensuring that $w_{ij}(q+2)$ is the maximum of the two weights. If we solve for $w_{ij}(q+2)$, we obtain

$$w_{ij}(q+2) = (1-\gamma)^2 w_{ij}(q) + \alpha.$$

Assuming that w_{ij} will eventually reach a steady state value, we can find it by setting both $w_{ij}(q+2)$ and $w_{ij}(q)$ equal to w_{ij}^{max} and solving

$$w_{ij}^{max} = (1-\gamma)^2 w_{ij}^{max} + \alpha,$$

$$w_{ij}^{max} = \frac{\alpha}{2\gamma - \gamma^2}.$$



We can use MATLAB to make a plot of this relationship. The plot will show learning rates and decay rates at intervals of 0.025.

```
lr = 0:0.025:1;
dr = 0.025:0.025:1;
```

Here are the commands for creating a mesh plot of the maximum weight, as a function of the learning and decay rate values.

```
[LR,DR] = meshgrid(dr,lr);
MW = LR ./ (DR .* (2 - DR));
mesh(DR,LR,MW);
```

The plot shows that w_{ij}^{max} approaches infinity as the decay rate γ becomes small with respect to the learning rate α (see Figure P15.1).

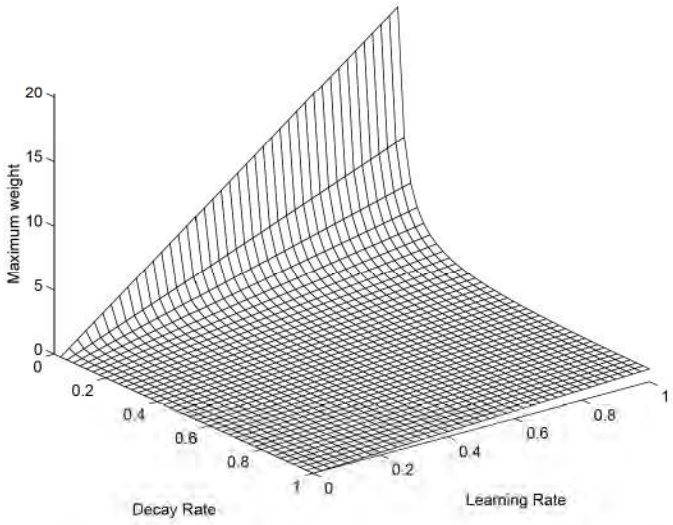


Figure P15.1 Maximum Weight w_{ij}^{max}

P15.2 Retrain the orange recognition network on page 13-13 using the instar rule with a learning rate of 0.4. Use the same training sequence. How many time steps are required for the network to learn to recognize an orange by its measurements?

Here is the training sequence. It is to be repeated until the network can respond to the orange measurements ($\mathbf{p} = \begin{bmatrix} 1 & -1 & -1 \end{bmatrix}^T$), even when the visual detection system fails ($p^0 = 0$).

$$\left\{ p^0(1) = 0, \mathbf{p}(1) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \left\{ p^0(2) = 1, \mathbf{p}(2) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \ldots$$

```
» 2 + 2
ans =
     4
```

We will use MATLAB to make the calculations. These two lines of code set the weights to their initial values.

```
w0 = 3;
W = [0 0 0];
```

We can then simulate the first time step of the network.

```
p0 = 0;
p = [1; -1; -1];
a = hardlim(w0*p0+W*p-2)
a =
     0
```

The neuron does not yet recognize the orange, so its output is 0. The weights do not change when updated with the instar rule.

```
W = W + 0.4*a*(p'-W)
W =
     0 0 0
```

The neuron begins learning the measurements in the second iteration.

```
p0 = 1;
p = [1; -1; -1];
a = hardlim(w0*p0+W*p-2)
a =
     1

W = W + 0.4*a*(p'-W)
W =
     0.4000 -0.4000 -0.4000
```

But the association is still not strong enough for a response in the third iteration.

```
p0 = 0;
p = [1; -1; -1];
a = hardlim(w0*p0+W*p-2)
a =
     0

W = W + 0.4*a*(p'-W)
W =
     0.4000 -0.4000 -0.4000
```

Here are the results of the fourth iteration:

```
a =
     1
W =
     0.6400 -0.6400 -0.6400
```

the fifth iteration:

```
a =
     0
W =
     0.6400 -0.6400 -0.6400
```

and the sixth iteration:

```
a =
     1
W =
     0.7840 -0.7840 -0.7840.
```

By the seventh iteration the network is able to recognize the orange by its measurements alone.

```
p0 = 0;
p = [1; -1; -1];
a = hardlim(w0*p0+W*p-2)
a =
     1

W = W + 0.4*a*(p'-W)
W =
     0.8704 -0.8704 -0.8704
```

Due to the lower learning rate, the network had to experience the measurements paired with its response three times (the even numbered iterations) before it developed a strong association between them.

P15.3 Both the recognition and recall networks used in this chapter's examples could only learn a single vector. Draw the diagram and determine the parameters of a network capable of recognizing and responding to the following two vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 5 \\ -5 \\ 5 \end{bmatrix} \qquad \mathbf{p}_2 = \begin{bmatrix} -5 \\ 5 \\ 5 \end{bmatrix}.$$

The network should only respond when an input vector is identical to one of these vectors.

We know the network must have three inputs, because it must recognize three-element vectors. We also know that it will have two outputs, one output for each response.

Such a network can be obtained by combining two instars into a single layer, as in Figure P15.2.

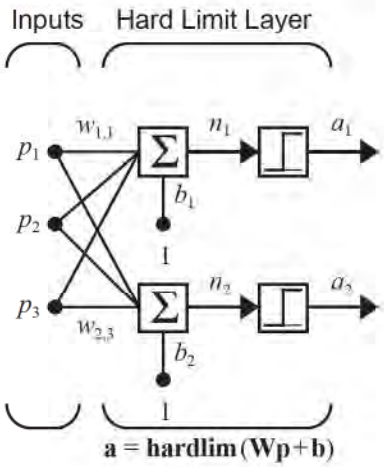


Figure P15.2 Two-Vector Recognition Network

We now set the weights ${}_1\mathbf{w}$ of the first neuron equal to $\mathbf{p}_1$, so that its net input will be at a maximum when an input vector points in the same direction as $\mathbf{p}_1$. Likewise, we will set ${}_2\mathbf{w}$ to $\mathbf{p}_2$ so that the second neuron is most sensitive to vectors in the direction of $\mathbf{p}_2$.

Combining the weight vectors gives us the weight matrix

$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \end{bmatrix} = \begin{bmatrix} 5 & -5 & 5 \\ -5 & 5 & 5 \end{bmatrix}.$$

(Note that this is the same manner in which we determined the weight matrix for the first layer of the Hamming network. In fact, the first layer of the Hamming network is a layer of instars. More about that in the next chapter.)

The lengths of $\mathbf{p}_1$ and $\mathbf{p}_2$ are the same:

$$\|\mathbf{p}_1\| = \|\mathbf{p}_2\| = \sqrt{(5)^2 + (-5)^2 + (5)^2} = \sqrt{75}.$$

To ensure that only an exact match between an input vector and a stored vector results in a response, both biases are set as follows (Eq. (15.29)):

$$b_1 = b_2 = -\|\mathbf{p}_1\|^2 = -75.$$

```
» 2 + 2
ans =
     4
```

We can use MATLAB to test that the network does indeed respond to $\mathbf{p}_1$.

```
W = [5 -5 5; -5 5 5];
b = [-75; -75];
```

```
p1 = [5; -5; 5];
a = hardlim(W*p1+b)
a =
     1
     0
```

The first neuron responded, indicating that the input vector was $\mathbf{p}_1$. The second neuron did not respond, because the input was not $\mathbf{p}_2$.

We can also check that the network does not respond to a third vector $\mathbf{p}_3$ that is not equal to either of the stored vectors.

```
p3 = [-5; 5; -5];
a = hardlim(W*p3+b)
a =
     0

     0
```

Neither neuron recognizes this new vector, so they both output 0.

P15.4 A single instar is being used for pattern recognition. Its weights and bias have the following values:

$$\mathbf{W} = {}_1\mathbf{w}^T = \begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \qquad b = -2 .$$

How close must an input vector (with a magnitude of $\sqrt{3}$) be to the weight vector for the neuron to output a 1? Find a vector that occurs on the border between those vectors that are recognized and those vectors that are not.

We begin by writing the expression for the neuron's output.

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b)$$

According to the definition of *hardlim*, a will be 1 if and only if the inner product between ${}_1\mathbf{w}^T$ and $\mathbf{p}$ is greater than or equal to $-b$ (Eq. (15.28)):

$${}_1\mathbf{w}^T\mathbf{p} = \|{}_1\mathbf{w}\|\|\mathbf{p}\|\cos\theta \geq -b .$$

We can find the maximum angle between ${}_1\mathbf{w}$ and $\mathbf{p}$ that meets this condition by substituting for the norms and solving

$$(\sqrt{3})(\sqrt{3})\cos\theta \geq 2$$

$$\theta \leq \cos^{-1}\left(\frac{2}{3}\right) = 48.19^\circ .$$

To find a borderline vector with magnitude $\sqrt{3}$, we need a vector $\mathbf{p}$ that meets the following conditions:

$$\|\mathbf{p}_1\| = \sqrt{p_1^2 + p_2^2 + p_3^2} = \sqrt{3},$$

$${}_1\mathbf{w}^T\mathbf{p} = w_1 p_1 + w_2 p_2 + w_3 p_3 - b = p_1 - p_2 - p_3 - 2 = 0.$$

Since we have three variables and only two constraints, we can set the third variable p_1 to 0 and solve

$$\sqrt{p_1^2 + p_2^2 + p_3^2} = \sqrt{3} \quad \Rightarrow \quad p_2^2 + p_3^2 = 3,$$

$$p_1 - p_2 - p_3 - 2 \quad \Rightarrow \quad p_2 + p_3 = -2,$$

$$(p_2 + p_3)^2 = p_2^2 + p_3^2 + 2p_2p_3 = (-2)^2 = 4,$$

$$3 + 2p_2p_3 = 4 \quad \Rightarrow \quad p_2p_3 = 0.5,$$

$$p_2(p_2 + p_3) = p_2^2 + p_2p_3 = p_2^2 + 0.5 = p_2(-2) = -2p_2.$$

After a little work we find that there are two possible values for p_2:

$$p_2^2 + 2p_2 + 0.5 = 0,$$

$$p_2 = -1 \pm \sqrt{0.5}.$$

It turns out that if we pick p_2 to be one of these values, then p_3 will take on the other value.

$$p_2 + p_3 = -1 \pm \sqrt{0.5} + p_3 = -2$$

$$p_3 = -1 \mp \sqrt{0.5}$$

Therefore, the following vector $\mathbf{p}$ is just the right distance from $\mathbf{w}$ to be recognized.

$$\mathbf{p} = \begin{bmatrix} 0 \\ -1 + \sqrt{0.5} \\ -1 - \sqrt{0.5} \end{bmatrix}$$

We can test it by presenting it to the network.

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b)$$

$$a = hardlim\left(\begin{bmatrix}1 & -1 & -1\end{bmatrix}\begin{bmatrix}0\\ -1+\sqrt{0.5}\\ -1-\sqrt{0.5}\end{bmatrix} - 2\right)$$

$$a = hardlim(0) = 1$$

The vector **p** does indeed result in a net input of 0 and is therefore on the boundary of the instar's active region.

P15.5 Consider the instar network shown in Figure P15.3. The training sequence for this network will consist of the following inputs:

$$\left\{p^0(1) = 0, \mathbf{p}(1) = \begin{bmatrix}-1\\ 1\end{bmatrix}\right\}, \left\{p^0(2) = 1, \mathbf{p}(2) = \begin{bmatrix}-1\\ 1\end{bmatrix}\right\}, \ldots .$$

These two sets of inputs are repeatedly presented to the network until the weight matrix W converges.

i. **Perform the first four iterations of the instar rule, with learning rate $\alpha = 0.5$. Assume that the initial W matrix is set to all zeros.**

ii. **Display the results of each iteration of the instar rule in graphical form (as in Figure 15.5).**

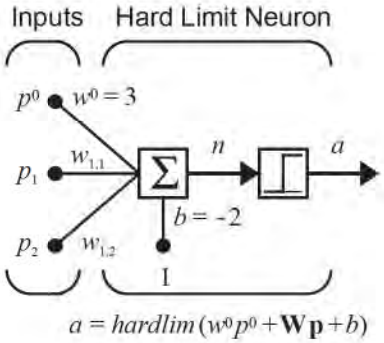


Figure P15.3 Instar Network for Problem P15.5

i. Because **W** initially contains all zeros, the instar does not respond to the measurements in the first iteration.

$$a(1) = hardlim(w^0 p^0(1) + \mathbf{W}\mathbf{p}(1) - 2)$$

$$a(1) = hardlim\left(3 \cdot 0 + \begin{bmatrix} 0 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \end{bmatrix} - 2\right) = 0$$

The neuron did not respond. Therefore its weights ${}_1\mathbf{w}$ are not altered by the instar rule.

$$\begin{aligned} {}_1\mathbf{w}(1) &= {}_1\mathbf{w}(0) + 0.5a(1)(\mathbf{p}(1) - {}_1\mathbf{w}(0)) \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned}$$

Because the unconditioned stimulus appears on the second iteration, the instar does respond.

$$a(2) = hardlim(w^0 p^0(2) + \mathbf{W}\mathbf{p}(2) - 2)$$

$$a(2) = hardlim\left(3 \cdot 1 + \begin{bmatrix} 0 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \end{bmatrix} - 2\right) = 1$$

The neuron did respond, and its weights ${}_1\mathbf{w}$ are updated by the instar rule.

$$\begin{aligned} {}_1\mathbf{w}(2) &= {}_1\mathbf{w}(1) + 0.5a(2)(\mathbf{p}(2) - {}_1\mathbf{w}(1)) \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0.5\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \end{aligned}$$

On the third iteration, the unconditioned stimulus is not presented, and the weights have not yet converged close enough to the input pattern. Therefore, the instar does not respond.

$$a(3) = hardlim(w^0 p^0(3) + \mathbf{W}\mathbf{p}(3) - 2)$$

$$a(3) = hardlim\left(3 \cdot 0 + \begin{bmatrix} -0.5 & 0.5 \end{bmatrix}\begin{bmatrix} -1 \\ 1 \end{bmatrix} - 2\right) = 0$$

Since the neuron did not respond, its weights are not updated.

$$
{}_1\mathbf{w}(3) = {}_1\mathbf{w}(2) + 0.5a(3)(\mathbf{p}(3) - {}_1\mathbf{w}(2))
$$

$$
= \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} + 0\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \right) = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}
$$

Because the unconditioned stimulus again appears on the fourth iteration, the instar does respond.

$$
a(4) = hardlim(w^0 p^0(4) + \mathbf{W}\mathbf{p}(4) - 2)
$$

$$
a(4) = hardlim\left(3 \cdot 1 + \begin{bmatrix} -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} - 2 \right) = 1
$$

Since the instar was activated, its weights are updated.

$$
{}_1\mathbf{w}(4) = {}_1\mathbf{w}(3) + 0.5a(4)(\mathbf{p}(4) - {}_1\mathbf{w}(3))
$$

$$
= \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} + 0.5\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \right) = \begin{bmatrix} -0.75 \\ 0.75 \end{bmatrix}
$$

This completes the fourth iteration. If we continue this process, ${}_1\mathbf{w}$ will converge to $\mathbf{p}$.

ii. Note that the weights are only updated (instar active) on iterations 2 and 4. Recall from Eq. (15.34) that when the instar is active, the learning rule can be written

$$
{}_1\mathbf{w}(q) = {}_1\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_1\mathbf{w}(q-1)) = (1-\alpha){}_1\mathbf{w}(q-1) + \alpha\mathbf{p}(q).
$$

When the instar is active, the weight vector is moved toward the input vector along a line between the old weight vector and the input vector. Figure P15.4 displays the movement of the weight vector for this problem. The weights were updated on iterations 2 and 4. Because $\alpha = 0.5$, whenever the instar is active the weight vector moves halfway from its current position toward the input vector.

$$
{}_1\mathbf{w}(q) = (0.5){}_1\mathbf{w}(q-1) + (0.5)\mathbf{p}(q)
$$

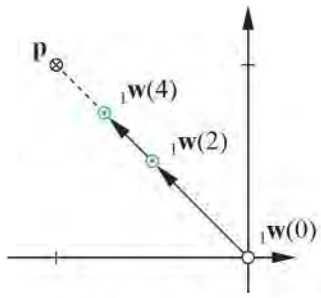


Figure P15.4 Instar Rule Example

Epilogue

In this chapter we introduced some simple networks capable of forming associations. We also developed and studied several learning rules that allowed networks to create new associations. Each rule operated by strengthening an association between any stimulus and response that occurred simultaneously.

The simple associative networks and learning rules developed in this chapter are useful in themselves, but they are also important building blocks for more powerful networks. In this chapter we introduced two networks, and associated learning rules, that will be fundamental for the development of important networks in the next three chapters: the instar and the outstar. The instar is a network that is trained to *recognize* a pattern. The outstar is a network that is trained to *recall* a pattern. We will use layers of instars in Chapters 16 and 18 to perform pattern recognition. These networks are very similar to the Hamming network of Chapter 3, whose first layer was, in fact, a layer of instars. In Chapter 19 we will introduce a more complex network, which combines both instars and outstars in order to produce stable learning.

Further Reading

[Ande72] J. Anderson, "A simple neural network generating an interactive memory," *Mathematical Biosciences*, vol. 14, pp. 197–220, 1972.

Anderson has proposed a "linear associator" model for associative memory. The model was trained, using a generalization of the Hebb postulate, to learn an association between input and output vectors. The physiological plausibility of the network was emphasized. Kohonen published a closely related paper at the same time [Koho72], although the two researchers were working independently.

[Gros68] S. Grossberg, "Some physiological and biochemical consequences of psychological postulates," *Proceedings of the National Academy of Sciences*, vol. 60, pp. 758–765, 1968.

This article describes early mathematical models (nonlinear differential equations) of associative learning. It synthesizes psychological, mathematical and physiological ideas.

[Gros82] S. Grossberg, *Studies of Mind and Brain*, Boston: D. Reidel Publishing Co., 1982.

This book is a collection of Stephen Grossberg papers from the period 1968 through 1980. It covers many of the fundamental concepts which are used in later Grossberg networks, such as the Adaptive Resonance Theory networks.

[Hebb49] D. O. Hebb, *The Organization of Behavior*, New York: Wiley, 1949.

The main premise of this seminal book was that behavior could be explained by the action of neurons. In it, Hebb proposed one of the first learning laws, which postulated a mechanism for learning at the cellular level.

[Koho72] T. Kohonen, "Correlation matrix memories," *IEEE Transactions on Computers*, vol. 21, pp. 353–359, 1972.

Kohonen proposed a correlation matrix model for associative memory. The model was trained, using the outer product rule (also known as the Hebb rule), to learn an association between input and output vectors. The mathematical structure of the network was emphasized. Anderson published a closely related paper at the same time [Ande72], although the two researchers were working independently.

[Koho87] T. Kohonen, *Self-Organization and Associative Memory*, 2nd Ed., Berlin: Springer-Verlag, 1987.

This book introduces the Kohonen rule and several networks that use it. It provides a complete analysis of linear associative models and gives many extensions and examples.

[Leib90] D. Lieberman, *Learning, Behavior and Cognition*, Belmont, CA: Wadsworth, 1990.

Leiberman's text forms an excellent introduction to behavioral psychology. This field is of interest to anyone looking to model human (or animal) learning with neural networks.

Exercises

E15.1 The network shown in Figure E15.1 is to be trained using the Hebb rule with decay, using a learning rate α of 0.3 and a decay rate γ of 0.1.

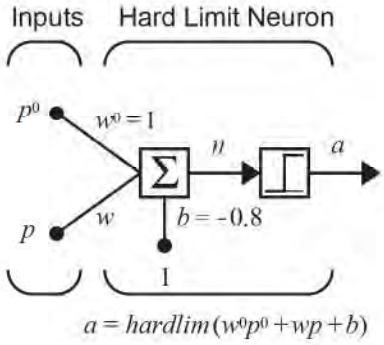


Figure E15.1 Associative Network

i. If w is initially set to 0, and w^0 and b remain constant (with the values shown in Figure E15.1), how many consecutive presentations of the following training set are required before the neuron will respond to the test set? Make a plot of w versus iteration number.

Training set: $\{p^0 = 1, p = 1\}$ Test set: $\{p^0 = 0, p = 1\}$

ii. Assume that w has an initial value of 1. How many consecutive presentations of the following training set are required before the neuron will no longer be able to respond to the test set? Make a plot of w versus iteration number.

Training set: $\{p^0 = 0, p = 0\}$ Test set: $\{p^0 = 0, p = 1\}$

E15.2 For Exercise E15.1 part (i), use Eq. (15.19) to determine the steady state value of w. Verify that this answer agrees with your plot from Exercise E15.1 part (i).

E15.3 Repeat Exercise E15.1, but this time use the Hebb rule without decay ($\gamma = 0$).

E15.4 The following rule looks similar to the instar rule, but it behaves quite differently:

$$\Delta w_{ij} = -\alpha a_i (p_j + w_{ij}^{old})$$

i. Determine the conditions under which the Δw_{ij} is nonzero.

ii. What value does the weight approach when Δw_{ij} is nonzero?

iii. Can you think of a use for this rule?

E15.5 The instar shown in Figure E15.2 is to be used to recognize a vector.

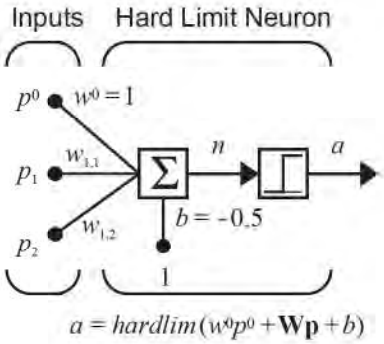


Figure E15.2 Vector Recognizer

i. Train the network with the instar rule on the following training sequence. Apply the instar rule to the second input's weights only (which should be initialized to zeros), using a learning rate of 0.6. The other weight and the bias are to remain constant at the values in the figure. (You may wish to use MATLAB to perform the calculations.)



$$\left\{p^0(1) = 1, \mathbf{p}(1) = \begin{bmatrix} 0.174 \\ 0.985 \end{bmatrix}\right\}\left\{p^0(2) = 0, \mathbf{p}(2) = \begin{bmatrix} -0.174 \\ 0.985 \end{bmatrix}\right\}$$

$$\left\{p^0(3) = 1, \mathbf{p}(3) = \begin{bmatrix} 0.174 \\ 0.985 \end{bmatrix}\right\}\left\{p^0(4) = 0, \mathbf{p}(4) = \begin{bmatrix} -0.174 \\ 0.985 \end{bmatrix}\right\}$$

$$\left\{p^0(5) = 1, \mathbf{p}(5) = \begin{bmatrix} 0.174 \\ 0.985 \end{bmatrix}\right\}\left\{p^0(6) = 0, \mathbf{p}(6) = \begin{bmatrix} -0.174 \\ 0.985 \end{bmatrix}\right\}$$

ii. What were your final values for $\mathbf{W}$?

iii. How do these final values compare with the vectors in the training sequence?

iv. What magnitude would you expect the weights to have after training, if the network were trained for many more iterations of the same training sequence?

E15.6 Consider the instar network shown in Figure E15.3. The training sequence for this network will consist of the following inputs:

$$\left\{p^0(1) = 0, \mathbf{p}(1) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right\}, \left\{p^0(2) = 1, \mathbf{p}(2) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right\}, \ldots .$$

These two sets of inputs are repeatedly presented to the network until the weight matrix $\mathbf{W}$ converges.

i. Perform the first eight iterations of the instar rule, with learning rate $\alpha = 0.25$. Assume that the initial $\mathbf{W}$ matrix is set to

$$\mathbf{W} = \begin{bmatrix} 1 & 0 \end{bmatrix}.$$

ii. Display the results of each iteration of the instar rule in graphical form (as in Figure 15.5).

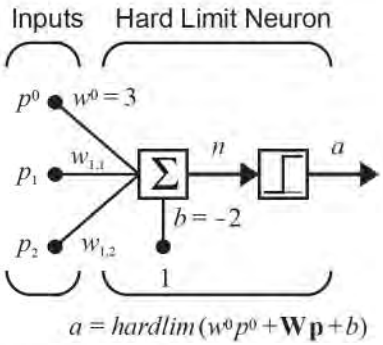


Figure E15.3 Instar Network for Exercise E15.6

E15.7 Draw a diagram of a network capable of recognizing three different four-element vectors (of ±1 values) when given different stimuli (of value 1).

i. How many inputs does your network have? How many outputs? What transfer function did you use?

ii. Choose values for the network's weights so that it can recognize each of the following vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \quad \mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

iii. Choose an appropriate value for the biases. Explain your choice.

iv. Test the network with one of the vectors above. Was its response correct?

v. Test the network with the following vector.

$$\mathbf{p}_1 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

Why did it respond the way it did?

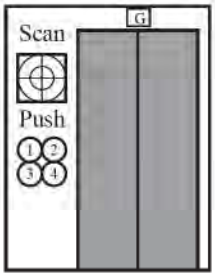
E15.8 This chapter included an example of a recognition network that initially used a visual system to identify oranges. At first the network needed the visual system to tell it when an orange was present, but eventually it learned to recognize oranges from sensor measurements.

i. Let us replace the visual system with a person. Initially, the network would depend on a person to tell it when an orange was present. Would you consider the network to be learning in a supervised or unsupervised manner?

ii. In what ways would the input from a person resemble the targets used to train supervised networks in earlier chapters?

iii. In what ways would it differ?

E15.9 The network shown in Figure E15.4 is installed in an elevator used by three senior executives in a plush high-security corporate building. It has buttons marked '1' through '4' for four floors above the ground floor. When an executive enters the elevator on the ground floor, it determines which person it is with a retinal scan, and then uses the network to select the floor where that person is most likely to go to. If the guess is incorrect, the person can push a different button at any time, but if the network is correct, it will save an important executive the effort of pushing a button.



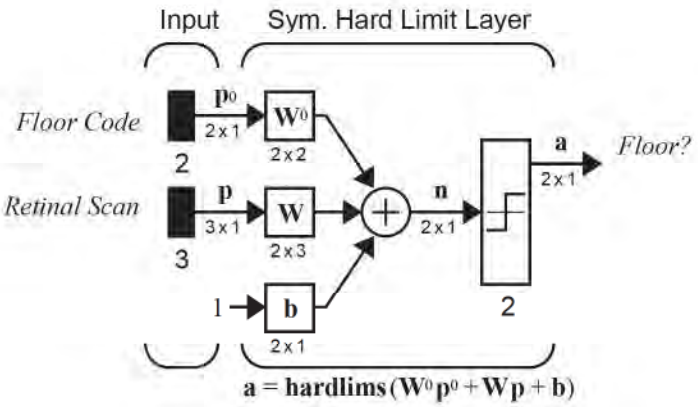
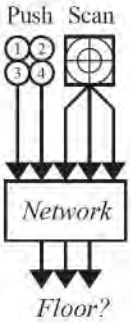


Figure E15.4 Elevator Network



The network's input/output function is

$$\mathbf{a} = \mathbf{hardlims}(\mathbf{W}^0\mathbf{p}^0 + \mathbf{W}\mathbf{p} + \mathbf{b}).$$

The first input $\mathbf{p}^0$ provides the network with a floor code, if a button has been pushed.

$$\mathbf{p}_1^0 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \text{ (1st floor)} \qquad \mathbf{p}_2^0 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ (2nd floor)}$$

$$\mathbf{p}_3^0 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \text{ (3rd floor)} \qquad \mathbf{p}_4^0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ (4th floor)}$$

If no button is pushed, then no code is given.

$$\mathbf{p}_0^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ (no button pushed)}$$

The first input is weighted with an identity matrix, and the biases are set to -0.5, so that if a button is pushed the network will respond with its code.

$$\mathbf{W}^0 = \mathbf{I}, \ \mathbf{b} = \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix}$$

The second input is always available. It consists of three elements that represent the three executives:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ (President)}, \quad \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ (Vice-President)}, \quad \mathbf{p}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ (Chairman)}.$$

The network learns to recall the executives' favorite floors by updating the second set of weights using the outstar rule (using a learning rate of 0.6). Initially those weights are set to zero:

$$\mathbf{W} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$



i. Use MATLAB to simulate the network for the following sequence of events:

President pushes '4', Vice-President pushes '3',
Chairman pushes '1', Vice-President pushes '3',
Chairman pushes '2', President pushes '4'.

In other words, train the network on the following sequence:

$$\{\mathbf{p}^0 = \mathbf{p}_4^0, \mathbf{p} = \mathbf{p}_1\}, \{\mathbf{p}^0 = \mathbf{p}_3^0, \mathbf{p} = \mathbf{p}_2\}, \{\mathbf{p}^0 = \mathbf{p}_1^0, \mathbf{p} = \mathbf{p}_3\},$$

$$\{\mathbf{p}^0 = \mathbf{p}_3^0, \mathbf{p} = \mathbf{p}_2\}, \{\mathbf{p}^0 = \mathbf{p}_2^0, \mathbf{p} = \mathbf{p}_3\}, \{\mathbf{p}^0 = \mathbf{p}_4^0, \mathbf{p} = \mathbf{p}_1\}.$$

ii. What are the final weights?

iii. Now continue simulating the network on these events:

President does not push a button,
Vice-President does not push a button,
Chairman does not push a button.

iv. Which floors did the network take each executive to?

v. If the executives were to push the following buttons many times, what would you expect the resulting weight matrix to look like?

President pushes '3',
Vice-President pushes '2',
Chairman pushes '4'.

16 Competitive Networks

Objectives	16-1
Theory and Examples	16-2
Hamming Network	16-3
Layer 1	16-3
Layer 2	16-4
Competitive Layer	16-5
Competitive Learning	16-7
Problems with Competitive Layers	16-9
Competitive Layers in Biology	16-10
Self-Organizing Feature Maps	16-12
Improving Feature Maps	16-15
Learning Vector Quantization	16-16
LVQ Learning	16-18
Improving LVQ Networks (LVQ2)	16-21
Summary of Results	16-22
Solved Problems	16-24
Epilogue	16-37
Further Reading	16-38
Exercises	16-39

Objectives

The Hamming network, introduced in Chapter 3, demonstrated one technique for using a neural network for pattern recognition. It required that the prototype patterns be known beforehand and incorporated into the network as rows of a weight matrix.

In this chapter we will discuss networks that are very similar in structure and operation to the Hamming network. Unlike the Hamming network, however, they use the associative learning rules of Chapter 15 to adaptively learn to classify patterns. Three such networks are introduced in this chapter: the competitive network, the feature map and the learning vector quantization (LVQ) network.

Theory and Examples

The Hamming network is one of the simplest examples of a competitive network. The neurons in the output layer of the Hamming network compete with each other to determine a winner. The winner indicates which prototype pattern is most representative of the input pattern. The competition is implemented by lateral inhibition — a set of negative connections between the neurons in the output layer. In this chapter we will illustrate how this competition can be combined with the associative learning rules of Chapter 15 to produce powerful self-organizing (unsupervised) networks.

As early as 1959, Frank Rosenblatt created a simple "spontaneous" classifier, an unsupervised network based on the perceptron, which learned to classify input vectors into two classes with roughly equal members.

In the late 1960s and early 1970s, Stephen Grossberg introduced many competitive networks that used lateral inhibition to good effect. Some of the useful behaviors he obtained were noise suppression, contrast-enhancement and vector normalization. His networks will be examined in Chapters 18 and 19.

In 1973, Christoph von der Malsburg introduced a self-organizing learning rule that allowed a network to classify inputs in such a way that neighboring neurons responded to similar inputs. The topology of his network mimicked, in some ways, the structures previously found in the visual cortex of cats by David Hubel and Torten Wiesel. His learning rule generated a great deal of interest, but it used a nonlocal calculation to ensure that weights were normalized. This made it less biologically plausible.

Grossberg extended von der Malsburg's work by rediscovering the instar rule, examined in Chapter 15. (The instar rule had previously been introduced by Nils Nilsson in his 1965 book *Learning Machines*.) Grossberg showed that the instar rule removed the necessity of re-normalizing weights, since weight vectors that learn to recognize normalized input vectors will automatically be normalized themselves.

The work of Grossberg and von der Malsburg emphasizes the biological plausibility of their networks. Another influential researcher, Teuvo Kohonen, has also been a strong proponent of competitive networks. However, his emphasis has been on engineering applications and efficient mathematical descriptions of the networks. During the 1970s he developed a simplified version of the instar rule and also, inspired by the work of von der Malsburg and Grossberg, found an efficient way to incorporate topology into a competitive network.

In this chapter we will concentrate on the Kohonen framework for competitive networks. His models illustrate the major features of competitive net-

works, and yet they are mathematically more tractable than the Grossberg networks. They provide a good introduction to competitive learning.

We will begin with the simple competitive network. Next we will present the *self-organizing feature map*, which incorporates a network topology. Finally, we will discuss *learning vector quantization*, which incorporates competition within a supervised learning framework.

Hamming Network

Since the competitive networks discussed in this chapter are closely related to the Hamming network (shown in Figure 16.1), it is worth reviewing the key concepts of that network first.

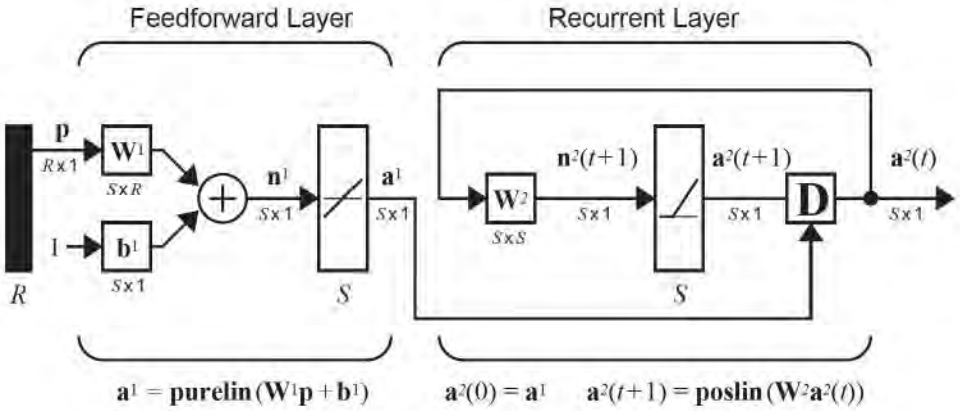


Figure 16.1 Hamming Network

The Hamming network consists of two layers. The first layer (which is a layer of instars) performs a correlation between the input vector and the prototype vectors. The second layer performs a competition to determine which of the prototype vectors is closest to the input vector.

Layer 1

Recall from Chapter 15 (see page 15-9 and following) that a single instar is able to recognize only one pattern. In order to allow multiple patterns to be classified, we need to have multiple instars. This is accomplished in the Hamming network.

Suppose that we want the network to recognize the following prototype vectors:

$$\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\}. \tag{16.1}$$

Then the weight matrix, $\mathbf{W}^1$, and the bias vector, $\mathbf{b}^1$, for Layer 1 will be:

$$\mathbf{W}^1 = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix}, \quad \mathbf{b}^1 = \begin{bmatrix} R \\ R \\ \vdots \\ R \end{bmatrix}, \tag{16.2}$$

where each row of $\mathbf{W}^1$ represents a prototype vector which we want to recognize, and each element of $\mathbf{b}^1$ is set equal to the number of elements in each input vector (R). (The number of neurons, S, is equal to the number of prototype vectors which are to be recognized, Q.)

Thus, the output of the first layer is

$$\mathbf{a}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1 = \begin{bmatrix} \mathbf{p}_1^T\mathbf{p} + R \\ \mathbf{p}_2^T\mathbf{p} + R \\ \vdots \\ \mathbf{p}_Q^T\mathbf{p} + R \end{bmatrix}. \tag{16.3}$$

Note that the outputs of Layer 1 are equal to the inner products of the prototype vectors with the input, plus R. As we discussed in Chapter 3 (page 3-9), these inner products indicate how close each of the prototype patterns is to the input vector. (This was also discussed in our presentation of the instar on page 15-10.)

Layer 2

In the instar of Chapter 15, a *hardlim* transfer function was used to decide if the input vector was close enough to the prototype vector. In Layer 2 of the Hamming network we have multiple instars, therefore we want to decide which prototype vector is closest to the input. Instead of the *hardlim* transfer function, we will use a competitive layer to choose the closest prototype.

Layer 2 is a competitive layer. The neurons in this layer are initialized with the outputs of the feedforward layer, which indicate the correlation between the prototype patterns and the input vector. Then the neurons compete with each other to determine a winner. After the competition, only one neuron will have a nonzero output. The winning neuron indicates which category of input was presented to the network (each prototype vector represents a category).

The first-layer output $\mathbf{a}^1$ is used to initialize the second layer.

$$\mathbf{a}^2(0) = \mathbf{a}^1 \tag{16.4}$$

Then the second-layer output is updated according to the following recurrence relation:

$$\mathbf{a}^2(t+1) = \mathbf{poslin}(\mathbf{W}^2\mathbf{a}^2(t)). \tag{16.5}$$

The second-layer weights $\mathbf{W}^2$ are set so that the diagonal elements are 1, and the off-diagonal elements have a small negative value.

$$w_{ij}^2 = \begin{cases} 1, & \text{if } i = j \\ -\varepsilon, & \text{otherwise} \end{cases}, \text{ where } 0 < \varepsilon < \frac{1}{S-1} \tag{16.6}$$

Lateral Inhibition

This matrix produces *lateral inhibition*, in which the output of each neuron has an inhibitory effect on all of the other neurons. To illustrate this effect, substitute weight values of 1 and $-\varepsilon$ for the appropriate elements of $\mathbf{W}^2$, and rewrite Eq. (16.5) for a single neuron.

$$a_i^2(t+1) = poslin\left(a_i^2(t) - \varepsilon \sum_{j \neq i} a_j^2(t)\right) \tag{16.7}$$

At each iteration, each neuron's output will decrease in proportion to the sum of the other neurons' outputs (with a minimum output of 0). The output of the neuron with the largest initial condition will decrease more slowly than the outputs of the other neurons. Eventually that neuron will be the only one with a positive output. At this point the network has reached steady state. The index of the second-layer neuron with a stable positive output is the index of the prototype vector that best matched the input.

Winner-Take-All

This is called a *winner-take-all competition*, since only one neuron will have a nonzero output. In Chapter 18 we will discuss other types of competition.



You may wish to experiment with the Hamming network and the apple/orange classification problem. The Neural Network Design Demonstration Hamming Classification (`nnd3hamc`) *was previously introduced in Chapter 3.*

Competitive Layer

Competition

The second-layer neurons in the Hamming network are said to be in *competition* because each neuron excites itself and inhibits all the other neurons. To simplify our discussions in the remainder of this chapter, we will define a transfer function that does the job of a recurrent competitive layer:

$$\mathbf{a} = \mathbf{compet}(\mathbf{n}). \tag{16.8}$$

It works by finding the index i^* of the neuron with the largest net input, and setting its output to 1 (with ties going to the neuron with the lowest index). All other outputs are set to 0.

$$a_i = \begin{cases} 1, i = i^* \\ 0, i \neq i^* \end{cases}, \text{ where } n_{i^*} \geq n_i, \forall i, \text{ and } i^* \leq i, \forall n_i = n_{i^*} \qquad (16.9)$$

Replacing the recurrent layer of the Hamming network with a competitive transfer function on the first layer will simplify our presentations in this chapter. (We will study the competition process in more detail in Chapter 18.) A competitive layer is displayed in Figure 16.2.

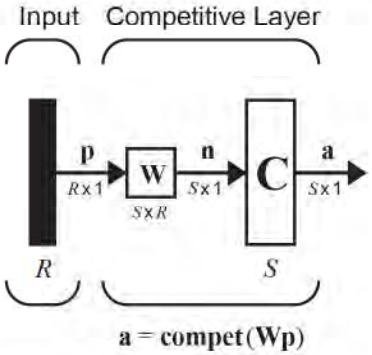


Figure 16.2 Competitive Layer

As with the Hamming network, the prototype vectors are stored in the rows of $\mathbf{W}$. The net input $\mathbf{n}$ calculates the distance between the input vector $\mathbf{p}$ and each prototype ${}_i\mathbf{w}$ (assuming vectors have normalized lengths of L). The net input n_i of each neuron i is proportional to the angle θ_i between $\mathbf{p}$ and the prototype vector ${}_i\mathbf{w}$:

$$\mathbf{n} = \mathbf{W}\mathbf{p} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix} \mathbf{p} = \begin{bmatrix} {}_1\mathbf{w}^T\mathbf{p} \\ {}_2\mathbf{w}^T\mathbf{p} \\ \vdots \\ {}_S\mathbf{w}^T\mathbf{p} \end{bmatrix} = \begin{bmatrix} L^2\cos\theta_1 \\ L^2\cos\theta_2 \\ \vdots \\ L^2\cos\theta_S \end{bmatrix}. \qquad (16.10)$$

The competitive transfer function assigns an output of 1 to the neuron whose weight vector points in the direction closest to the input vector:

$$\mathbf{a} = \mathbf{compet}(\mathbf{W}\mathbf{p}). \qquad (16.11)$$



To experiment with the competitive network and the apple/orange classification problem, use the Neural Network Design Demonstration Competitive Classification (nnd14cc).

Competitive Learning

We can now design a competitive network classifier by setting the rows of $\mathbf{W}$ to the desired prototype vectors. However, we would like to have a learning rule that could be used to train the weights in a competitive network, without knowing the prototype vectors. One such learning rule is the instar rule from Chapter 15:

$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha a_i(q)(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) . \tag{16.12}$$

For the competitive network, $\mathbf{a}$ is only nonzero for the winning neuron ($i = i^*$). Therefore, we can get the same results using the Kohonen rule.

$$\begin{aligned} {}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\ &= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \end{aligned} \tag{16.13}$$

and

$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) \qquad i \neq i^* \tag{16.14}$$

Thus, the row of the weight matrix that is closest to the input vector (or has the largest inner product with the input vector) moves toward the input vector. It moves along a line between the old row of the weight matrix and the input vector, as shown in Figure 16.3.

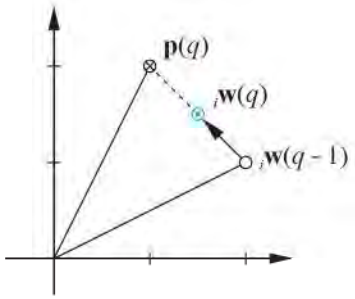


Figure 16.3 Graphical Representation of the Kohonen Rule



Let's use the six vectors in Figure 16.4 to demonstrate how a competitive layer learns to classify vectors. Here are the six vectors:

$$\begin{aligned} \mathbf{p}_1 &= \begin{bmatrix} -0.1961 \\ 0.9806 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 0.9806 \\ 0.1961 \end{bmatrix} \\ \mathbf{p}_4 &= \begin{bmatrix} 0.9806 \\ -0.1961 \end{bmatrix}, \mathbf{p}_5 = \begin{bmatrix} -0.5812 \\ -0.8137 \end{bmatrix}, \mathbf{p}_6 = \begin{bmatrix} -0.8137 \\ -0.5812 \end{bmatrix} . \end{aligned} \tag{16.15}$$

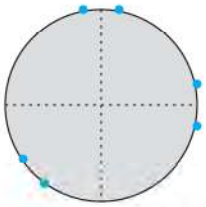
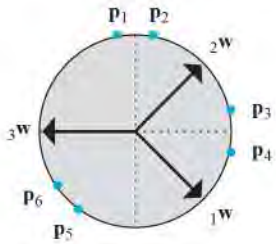


Figure 16.4 Sample Input Vectors

Our competitive network will have three neurons, and therefore it can classify vectors into three classes. Here are the "randomly" chosen normalized initial weights:

$$ {}_1\mathbf{w} = \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}, {}_2\mathbf{w} = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}, {}_3\mathbf{w} = \begin{bmatrix} -1.0000 \\ 0.0000 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ {}_3\mathbf{w}^T \end{bmatrix}. \quad (16.16) $$



The data vectors are shown at left, with the weight vectors displayed as arrows. Let's present the vector $\mathbf{p}_2$ to the network:

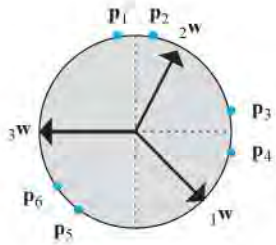
$$ \mathbf{a} = \mathbf{compet}(\mathbf{W}\mathbf{p}_2) = \mathbf{compet}\left(\begin{bmatrix} 0.7071 & -0.7071 \\ 0.7071 & 0.7071 \\ -1.0000 & 0.0000 \end{bmatrix}\begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix}\right) \quad (16.17) $$

$$ = \mathbf{compet}\left(\begin{bmatrix} -0.5547 \\ 0.8321 \\ -0.1961 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}. $$

The second neuron's weight vector was closest to $\mathbf{p}_2$, so it won the competition ($i^* = 2$) and output a 1. We now apply the Kohonen learning rule to the winning neuron with a learning rate of $\alpha = 0.5$.

$$ {}_2\mathbf{w}^{new} = {}_2\mathbf{w}^{old} + \alpha(\mathbf{p}_2 - {}_2\mathbf{w}^{old}) \quad (16.18) $$

$$ = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix} + 0.5\left(\begin{bmatrix} 0.1961 \\ 0.9806 \end{bmatrix} - \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}\right) = \begin{bmatrix} 0.4516 \\ 0.8438 \end{bmatrix} $$



The Kohonen rule moves ${}_2\mathbf{w}$ closer to $\mathbf{p}_2$, as can be seen in the diagram at left. If we continue choosing input vectors at random and presenting them to the network, then at each iteration the weight vector closest to the input vector will move toward that vector. Eventually, each weight vector will

point at a different cluster of input vectors. Each weight vector becomes a prototype for a different cluster.

This problem is simple enough that we can predict which weight vector will point at which cluster. The final weights will look something like those shown in Figure 16.5.

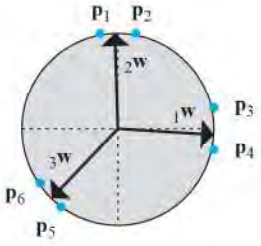
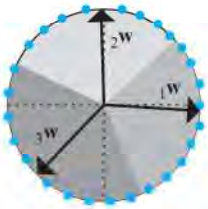


Figure 16.5 Final Weights

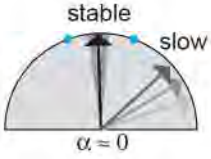


Once the network has learned to cluster the input vectors, it will classify new vectors accordingly.The diagram in the left margin uses shading to show which region each neuron will respond to. The competitive layer assigns each input vector $\mathbf{p}$ to one of these classes by producing an output of 1 for the neuron whose weight vector is closest to $\mathbf{p}$.

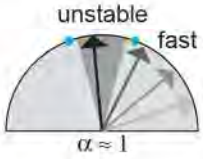


To experiment with the competitive learning use the Neural Network Design Demonstration Competitive Learning (`nnd14c1`).

Problems with Competitive Layers



Competitive layers make efficient adaptive classifiers, but they do suffer from a few problems. The first problem is that the choice of learning rate forces a trade-off between the speed of learning and the stability of the final weight vectors. A learning rate near zero results in slow learning. However, once a weight vector reaches the center of a cluster it will tend to stay close to the center.



In contrast, a learning rate near 1.0 results in fast learning. However, once the weight vector has reached a cluster, it will continue to oscillate as different vectors in the cluster are presented.

Sometimes this trade-off between fast learning and stability can be used to advantage. Initial training can be done with a large learning rate for fast learning. Then the learning rate can be decreased as training progresses, to achieve stable prototype vectors. Unfortunately, this technique will not work if the network needs to continuously adapt to new arrangements of input vectors.

A more serious stability problem occurs when clusters are close together. In certain cases, a weight vector forming a prototype of one cluster may "in-

vade" the territory of another weight vector, and therefore upset the current classification scheme.

The series of four diagrams in Figure 16.6 illustrate this problem. Two input vectors (shown with blue circles in diagram (a)) are presented several times. The result is that the weight vectors representing the middle and right clusters shift to the right. Eventually one of the right cluster vectors is reclassified by the center weight vector. Further presentations move the middle vector over to the right until it "loses" some of its vectors, which then become part of the class associated with the left weight vector.



Figure 16.6 Example of Unstable Learning

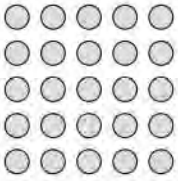


A third problem with competitive learning is that occasionally a neuron's initial weight vector is located so far from any input vectors that it never wins the competition, and therefore never learns. The result is a "dead" neuron, which does nothing useful. For example, the downward-pointing weight vector in the diagram to the left will never learn, regardless of the order in which vectors are presented. One solution to this problem consists of adding a negative bias to the net input of each neuron and then decreasing the bias each time the neuron wins. This will make it harder for a neuron to win the competition if it has won often. This mechanism is sometimes called a "conscience." (See Exercise E16.4.)

Finally, a competitive layer always has as many classes as it has neurons. This may not be acceptable for some applications, especially when the number of clusters is not known in advance. In addition, for competitive layers, each class consists of a convex region of the input space. Competitive layers cannot form classes with nonconvex regions or classes that are the union of unconnected regions.

Some of the problems discussed in this section are solved by the feature map and LVQ networks, which are introduced in later sections of this chapter, and the ART networks, which are presented in Chapter 19.

Competitive Layers in Biology

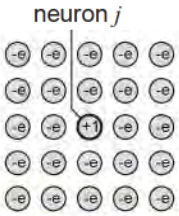


In previous chapters we have made no mention of how neurons are physically organized within a layer (the topology of the network). In biological neural networks, neurons are typically arranged in two-dimensional layers, in which they are densely interconnected through lateral feedback. The diagram to the left shows a layer of twenty-five neurons arranged in a two-dimensional grid.

Often weights vary as a function of the distance between the neurons they connect. For example, the weights for Layer 2 of the Hamming network are assigned as follows:

$$w_{ij} = \begin{cases} 1, & \text{if } i = j \\ -\varepsilon, & \text{if } i \neq j \end{cases} . \tag{16.19}$$

Eq. (16.20) assigns the same values as Eq. (16.19), but in terms of the distances d_{ij} between neurons:



$$w_{ij} = \begin{cases} 1, & \text{if } d_{ij} = 0 \\ -\varepsilon, & \text{if } d_{ij} > 0 \end{cases} . \tag{16.20}$$

Either Eq. (16.19) or Eq. (16.20) will assign the weight values shown in the diagram at left. Each neuron i is labeled with the value of the weight w_{ij}, which comes from it to the neuron marked j.

On-center/off-surround

The term *on-center/off-surround* is often used to describe such a connection pattern between neurons. Each neuron reinforces itself (center), while inhibiting all other neurons (surround).

It turns out that this is a crude approximation of biological competitive layers. In biology, a neuron reinforces not only itself, but also those neurons close to it. Typically, the transition from reinforcement to inhibition occurs smoothly as the distance between neurons increases.

This transition is illustrated on the left side of Figure 16.7. This is a function that relates the distance between neurons to the weight connecting them. Those neurons that are close provide excitatory (reinforcing) connections, and the magnitude of the excitation decreases as the distance increases. Beyond a certain distance, the neurons begin to have inhibitory connections, and the inhibition increases as the distance increases. Because of its shape, the function is referred to as the *Mexican-hat function*. On the right side of Figure 16.7 is a two-dimensional illustration of the Mexican-hat (on-center/off-surround) function. Each neuron i is marked to show the sign and relative strength of its weight w_{ij} going to neuron j.

Mexican-Hat Function



Figure 16.7 On-Center/Off-Surround Layer in Biology

Biological competitive systems, in addition to having a gradual transition between excitatory and inhibitory regions of the on-center/off-surround connection pattern, also have a weaker form of competition than the winner-take-all competition of the Hamming network. Instead of a single active neuron (winner), biological networks generally have "bubbles" of activity that are centered around the most active neuron. This is caused in part by the form of the on-center/off-surround connectivity pattern and also by nonlinear feedback connections. (See the discussion on contour enhancement in Chapter 18.)

Self-Organizing Feature Maps

In order to emulate the activity bubbles of biological systems, without having to implement the nonlinear on-center/off-surround feedback connections, Kohonen designed the following simplification. His self-organizing feature map (SOFM) network first determines the winning neuron i^* using the same procedure as the competitive layer. Next, the weight vectors for all neurons within a certain neighborhood of the winning neuron are updated using the Kohonen rule,

SOFM

$$\begin{aligned} {}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\ &= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \end{aligned} \qquad i \in N_{i^*}(d), \tag{16.21}$$

Neighborhood

where the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*:

$$N_i(d) = \{j, d_{ij} \le d\}. \tag{16.22}$$

When a vector $\mathbf{p}$ is presented, the weights of the winning neuron *and* its neighbors will move toward $\mathbf{p}$. The result is that, after many presentations, neighboring neurons will have learned vectors similar to each other.



To demonstrate the concept of a neighborhood, consider the two diagrams shown in Figure 16.8. The left diagram illustrates a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$.

The definition of these neighborhoods would be

$$N_{13}(1) = \{8, 12, 13, 14, 18\}, \tag{16.23}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}. \tag{16.24}$$

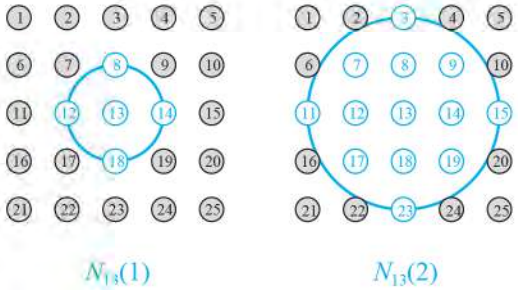


Figure 16.8 Neighborhoods

We should mention that the neurons in an SOFM do not have to be arranged in a two-dimensional pattern. It is possible to use a one-dimensional arrangement, or even three or more dimensions. For a one-dimensional SOFM, a neuron will only have two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). It is also possible to define distance in different ways. For instance, Kohonen has suggested rectangular and hexagonal neighborhoods for efficient implementation. The performance of the network is not sensitive to the exact shape of the neighborhoods.



Now let's demonstrate the performance of an SOFM network. Figure 16.9 shows a feature map and the two-dimensional topology of its neurons.

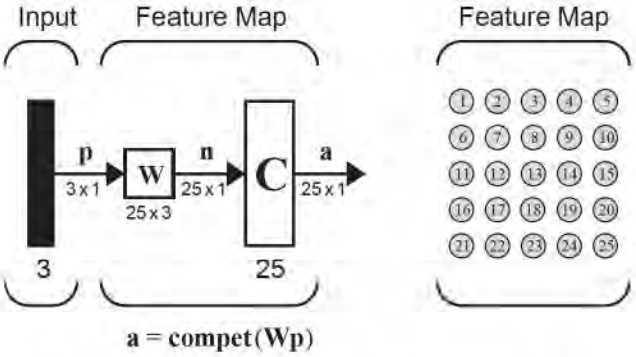
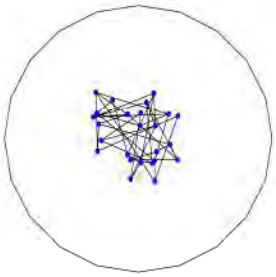
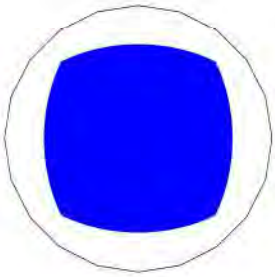


Figure 16.9 Self-Organizing Feature Map



The diagram in the left margin shows the initial weight vectors for the feature map. Each three-element weight vector is represented by a dot on the sphere. (The weights are normalized, therefore they will fall on the surface of a sphere.) Dots of neighboring neurons are connected by lines so you can see how the physical topology of the network is arranged in the input space.



The diagram to the left shows a square region on the surface of the sphere. We will randomly pick vectors in this region and present them to the feature map.

Each time a vector is presented, the neuron with the closest weight vector will win the competition. The winning neuron and its neighbors move their weight vectors closer to the input vector (and therefore to each other). For this example we are using a neighborhood with a radius of 1.

The weight vectors have two tendencies: first, they spread out over the input space as more vectors are presented; second, they move toward the weight vectors of neighboring neurons. These two tendencies work together to rearrange the neurons in the layer so that they evenly classify the input space.

The series of diagrams in Figure 16.10 shows how the weights of the twenty-five neurons spread out over the active input space and organize themselves to match its topology.

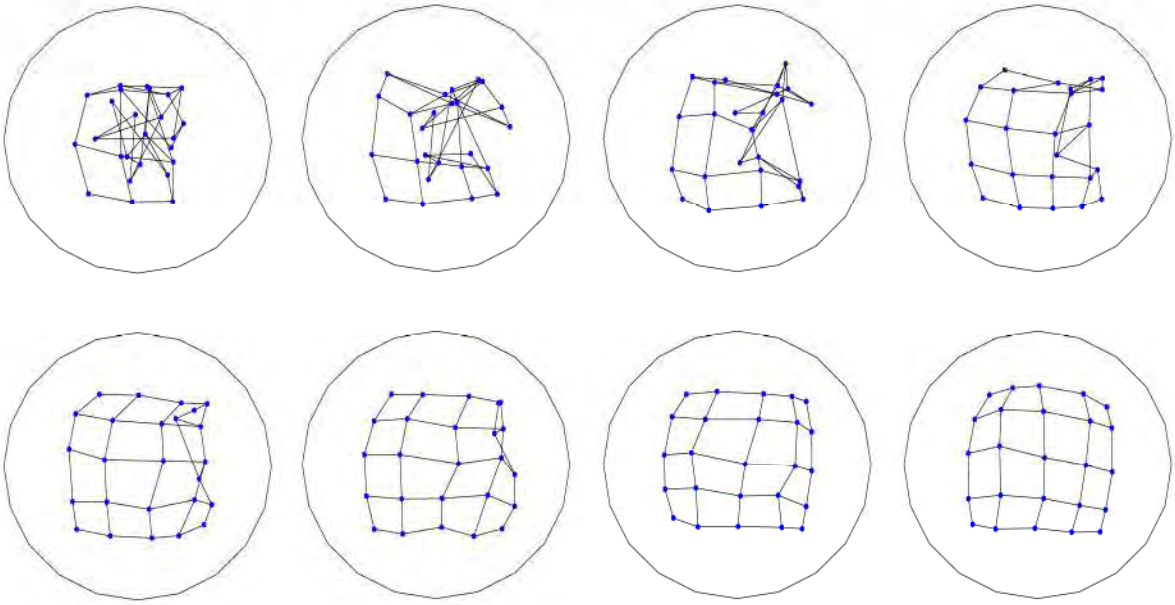


Figure 16.10 Self-Organization, 250 Iterations per Diagram

In this example, the input vectors were generated with equal probability from any point in the input space. Therefore, the neurons classify roughly equal areas of the input space.

Figure 16.11 provides more examples of input regions and the resulting feature maps after self-organization.

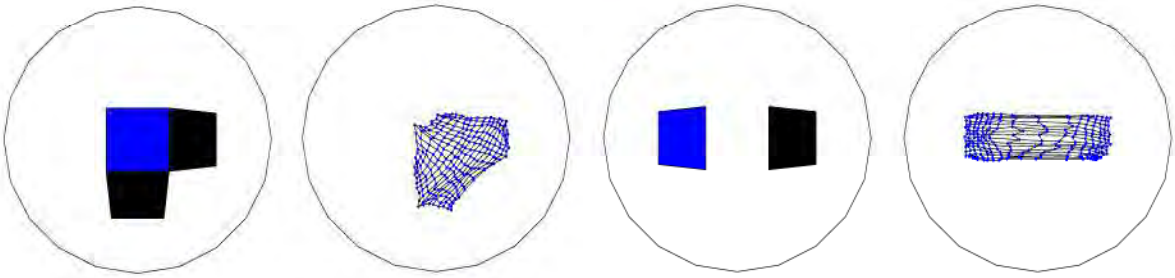


Figure 16.11 Other Examples of Feature Map Training

Occasionally feature maps can fail to properly fit the topology of their input space. This usually occurs when two parts of the net fit the topology of separate parts of the input space, but the net forms a twist between them. An example is given in Figure 16.12.

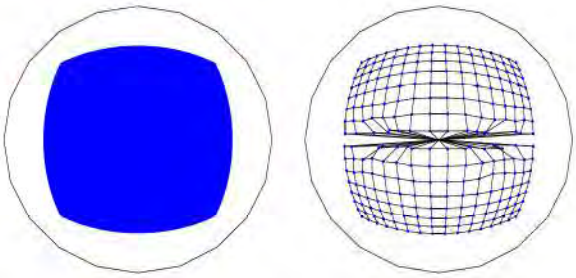


Figure 16.12 Feature Map with a Twist

It is unlikely that this twist will ever be removed, because the two ends of the net have formed stable classifications of different regions.

Improving Feature Maps

So far, we have described only the most basic algorithm for training feature maps. Now let's consider several techniques that can be used to speed up the self-organizing process and to make it more reliable.

One method to improve the performance of the feature map is to vary the size of the neighborhoods during training. Initially, the neighborhood size, d, is set large. As training progresses, d is gradually reduced, until it only includes the winning neuron. This speeds up self-organizing and makes twists in the map very unlikely.

The learning rate can also be varied over time. An initial rate of 1 allows neurons to quickly learn presented vectors. During training, the learning rate is decreased asymptotically toward 0, so that learning becomes stable.

(We discussed the use of this technique for competitive layers earlier in the chapter.)

Another alteration that speeds self-organization is to have the winning neuron use a larger learning rate than the neighboring neurons.

Finally, both competitive layers and feature maps often use an alternative expression for net input. Instead of using the inner product, they can directly compute the distance between the input vector and the prototype vectors. The advantage of using the distance is that input vectors do not need to be normalized. This alternative net input expression is introduced in the next section on LVQ networks.

Other enhancements to the SOFM are described in Chapter 26, including a batch version of the SOFM learning rule. That chapter is a case study of using the SOFM for clustering.



To experiment with feature maps use the Neural Network Design Demonstrations 1-D Feature Maps (`nnd14fm1`) *and 2-D Feature Maps* (`nnd14fm2`).

Learning Vector Quantization

The final network we will introduce in this chapter is the learning vector quantization (LVQ) network, which is shown in Figure 16.13. The LVQ network is a hybrid network. It uses both unsupervised and supervised learning to form classifications.

In the LVQ network, each neuron in the first layer is assigned to a class, with several neurons often assigned to the same class. Each class is then assigned to one neuron in the second layer. The number of neurons in the first layer, S^1, will therefore always be at least as large as the number of neurons in the second layer, S^2, and will usually be larger.

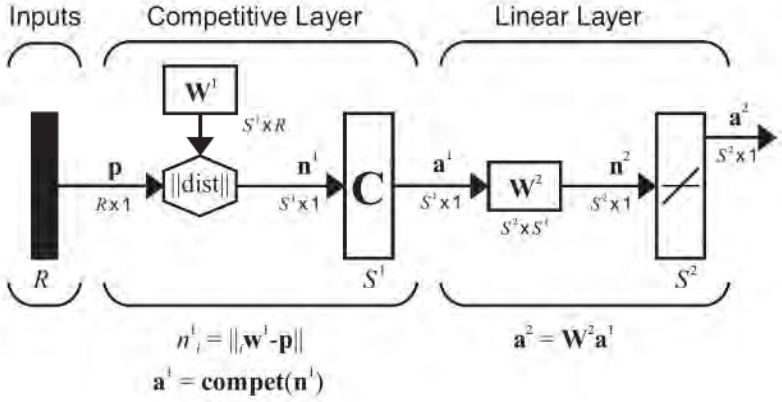


Figure 16.13 LVQ Network

As with the competitive network, each neuron in the first layer of the LVQ network learns a prototype vector, which allows it to classify a region of the input space. However, instead of computing the proximity of the input and weight vectors by using the inner product, we will simulate the LVQ networks by calculating the distance directly. One advantage of calculating the distance directly is that vectors need not be normalized. When the vectors are normalized, the response of the network will be the same, whether the inner product is used or the distance is directly calculated.

The net input of the first layer of the LVQ will be

$$n_i^1 = -\left\| {}_i\mathbf{w}^1 - \mathbf{p} \right\|, \tag{16.25}$$

or, in vector form,

$$\mathbf{n}^1 = -\begin{bmatrix} \left\| {}_1\mathbf{w}^1 - \mathbf{p} \right\| \\ \left\| {}_2\mathbf{w}^1 - \mathbf{p} \right\| \\ \vdots \\ \left\| {}_{S^1}\mathbf{w}^1 - \mathbf{p} \right\| \end{bmatrix}. \tag{16.26}$$

The output of the first layer of the LVQ is

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1). \tag{16.27}$$

Therefore the neuron whose weight vector is closest to the input vector will output a 1, and the other neurons will output 0.

Thus far, the LVQ network behaves exactly like the competitive network (at least for normalized vectors). There is a difference in interpretation, however. In the competitive network, the neuron with the nonzero output indicates which class the input vector belongs to. For the LVQ network, the winning neuron indicates a *subclass*, rather than a class. There may be several different neurons (subclasses) that make up each class.

Subclass

The second layer of the LVQ network is used to combine subclasses into a single class. This is done with the $\mathbf{W}^2$ matrix. The columns of $\mathbf{W}^2$ represent subclasses, and the rows represent classes. $\mathbf{W}^2$ has a single 1 in each column, with the other elements set to zero. The row in which the 1 occurs indicates which class the appropriate subclass belongs to.

$$(w_{ki}^2 = 1) \Rightarrow \text{subclass } i \text{ is a part of class } k \tag{16.28}$$

The process of combining subclasses to form a class allows the LVQ network to create complex class boundaries. A standard competitive layer has

the limitation that it can only create decision regions that are convex. The LVQ network overcomes this limitation.

LVQ Learning

The learning in the LVQ network combines competitive learning with supervision. As with all supervised learning algorithms, it requires a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}.$$

Each target vector must contain only zeros, except for a single 1. The row in which the 1 appears indicates the class to which the input vector belongs. For example, if we have a problem where we would like to classify a particular three-element vector into the second of four classes, we can express this as

$$\left\{\mathbf{p}_1 = \begin{bmatrix} \sqrt{1/2} \\ 0 \\ \sqrt{1/2} \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}\right\}. \tag{16.29}$$

Before learning can occur, each neuron in the first layer is assigned to an output neuron. This generates the matrix $\mathbf{W}^2$. Typically, equal numbers of hidden neurons are connected to each output neuron, so that each class can be made up of the same number of convex regions. All elements of $\mathbf{W}^2$ are set to zero, except for the following:

$$\text{If hidden neuron } i \text{ is to be assigned to class } k, \text{ then set } w^2_{ki} = 1. \tag{16.30}$$

Once $\mathbf{W}^2$ is defined, it will never be altered. The hidden weights $\mathbf{W}^1$ are trained with a variation of the Kohonen rule.

The LVQ learning rule proceeds as follows. At each iteration, an input vector $\mathbf{p}$ is presented to the network, and the distance from $\mathbf{p}$ to each prototype vector is computed. The hidden neurons compete, neuron i^* wins the competition, and the i^* th element of $\mathbf{a}^1$ is set to 1. Next, $\mathbf{a}^1$ is multiplied by $\mathbf{W}^2$ to get the final output $\mathbf{a}^2$, which also has only one nonzero element, k^*, indicating that $\mathbf{p}$ is being assigned to class k^*.

The Kohonen rule is used to improve the hidden layer of the LVQ network in two ways. First, if $\mathbf{p}$ is classified correctly, then we move the weights ${}_{i^*}\mathbf{w}^1$ of the winning hidden neuron toward $\mathbf{p}$.

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a^2_{k^*} = t_{k^*} = 1 \tag{16.31}$$

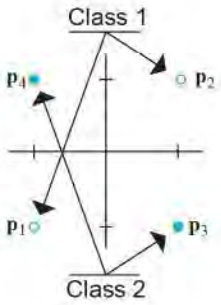
Second, if $\mathbf{p}$ was classified incorrectly, then we know that the wrong hidden neuron won the competition, and therefore we move its weights ${}_{i^*}\mathbf{w}^1$ *away* from $\mathbf{p}$.

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a_{k^*}^2 = 1 \neq t_{k^*} = 0 \quad (16.32)$$

The result will be that each hidden neuron moves toward vectors that fall into the class for which it forms a subclass and away from vectors that fall into other classes.

Let's take a look at an example of LVQ training. We would like to train an LVQ network to solve the following classification problem:

$$\text{class 1:}\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix}1\\1\end{bmatrix}\right\}, \text{ class 2:}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}\right\}, \quad (16.33)$$



as illustrated by the figure in the left margin. We begin by assigning target vectors to each input:

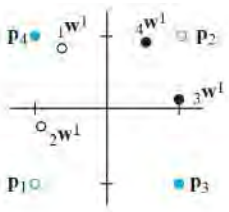
$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}1\\0\end{bmatrix}\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}1\\0\end{bmatrix}\right\}, \quad (16.34)$$

$$\left\{\mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}\right\}, \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\}. \quad (16.35)$$

We now must choose how many subclasses will make up each of the two classes. If we let each class be the union of two subclasses, we will end up with four neurons in the hidden layer. The output layer weight matrix will be

$$\mathbf{W}^2 = \begin{bmatrix}1 & 1 & 0 & 0\\0 & 0 & 1 & 1\end{bmatrix}. \quad (16.36)$$

$\mathbf{W}^2$ connects hidden neurons 1 and 2 to output neuron 1. It connects hidden neurons 3 and 4 to output neuron 2. Each class will be made up of two convex regions.



The row vectors in $\mathbf{W}^1$ are initially set to random values. They can be seen in the diagram at left. The weights belonging to the two hidden neurons that define class 1 are marked with hollow circles. The weights defining class 2 are marked with solid circles. The values for these weights are

$${}_1\mathbf{w}^1 = \begin{bmatrix}-0.543\\0.840\end{bmatrix}, {}_2\mathbf{w}^1 = \begin{bmatrix}-0.969\\-0.249\end{bmatrix}, {}_3\mathbf{w}^1 = \begin{bmatrix}0.997\\0.094\end{bmatrix}, {}_4\mathbf{w}^1 = \begin{bmatrix}0.456\\0.954\end{bmatrix}. \quad (16.37)$$

At each iteration of the training process, we present an input vector, find its response, and then adjust the weights. In this case we will begin by presenting $\mathbf{p}_3$.

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1) = \mathbf{compet}\left(\begin{bmatrix} -\|{}_1\mathbf{w}^1 - \mathbf{p}_3\| \\ -\|{}_2\mathbf{w}^1 - \mathbf{p}_3\| \\ -\|{}_3\mathbf{w}^1 - \mathbf{p}_3\| \\ -\|{}_4\mathbf{w}^1 - \mathbf{p}_3\| \end{bmatrix}\right) \tag{16.38}$$

$$= \mathbf{compet}\left(\begin{bmatrix} -\|[-0.543\ \ 0.840]^T - [1\ -1]^T\| \\ -\|[-0.969\ -0.249]^T - [1\ -1]^T\| \\ -\|[0.997\ \ 0.094]^T - [1\ -1]^T\| \\ -\|[0.456\ \ 0.954]^T - [1\ -1]^T\| \end{bmatrix}\right) = \mathbf{compet}\left(\begin{bmatrix} -2.40 \\ -2.11 \\ -1.09 \\ -2.03 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

The third hidden neuron has the closest weight vector to $\mathbf{p}_3$. In order to determine which class this neuron belongs to, we multiply $\mathbf{a}^1$ by $\mathbf{W}^2$.

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{16.39}$$

This output indicates that $\mathbf{p}_3$ is a member of class 2. This is correct, so ${}_3\mathbf{w}^1$ is updated by moving it toward $\mathbf{p}_3$.

$${}_3\mathbf{w}^1(1) = {}_3\mathbf{w}^1(0) + \alpha(\mathbf{p}_3 - {}_3\mathbf{w}^1(0)) \tag{16.40}$$

$$= \begin{bmatrix} 0.997 \\ 0.094 \end{bmatrix} + 0.5\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} - \begin{bmatrix} 0.997 \\ 0.094 \end{bmatrix}\right) = \begin{bmatrix} 0.998 \\ -0.453 \end{bmatrix}$$

The diagram on the left side of Figure 16.14 shows the weights after ${}_3\mathbf{w}^1$ was updated on the first iteration. The diagram on the right side of Figure 16.14 shows the weights after the algorithm has converged.

The diagram on the right side of Figure 16.14 also indicates how the regions of the input space will be classified. The regions that will be classified as class 1 are shown in gray, and the regions that will be classified as class 2 are shown in blue.

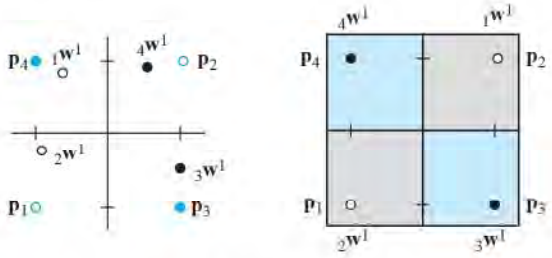


Figure 16.14 After First and Many Iterations

Improving LVQ Networks (LVQ2)

The LVQ network described above works well for many problems, but it does suffer from a couple of limitations. First, as with competitive layers, occasionally a hidden neuron in an LVQ network can have initial weight values that stop it from ever winning the competition. The result is a dead neuron that never does anything useful. This problem is solved with the use of a "conscience" mechanism, a technique discussed earlier for competitive layers, and also presented in Exercise E16.4.

Secondly, depending on how the initial weight vectors are arranged, a neuron's weight vector may have to travel through a region of a class that it doesn't represent, to get to a region that it does represent. Because the weights of such a neuron will be repulsed by vectors in the region it must cross, it may not be able to cross, and so it may never properly classify the region it is being attracted to. This is usually solved by applying the following modification to the Kohonen rule.

If the winning neuron in the hidden layer incorrectly classifies the current input, we move its weight vector away from the input vector, as before. However, we also adjust the weights of the closest neuron to the input vector that does classify it properly. The weights for this second neuron should be moved toward the input vector.

When the network correctly classifies an input vector, the weights of only one neuron are moved toward the input vector. However, if the input vector is incorrectly classified, the weights of two neurons are updated, one weight vector is moved away from the input vector, and the other one is moved toward the input vector. The resulting algorithm is called *LVQ2*.

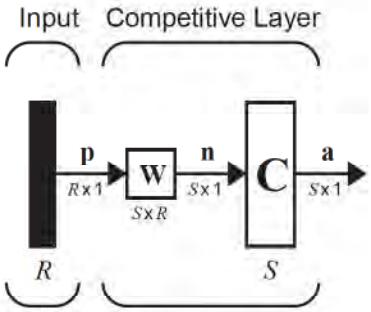
LVQ2



To experiment with LVQ networks use the Neural Network Design Demonstrations LVQ1 Networks (`nnd14lv1`) *and LVQ2 Networks* (`nnd14lv2`).

Summary of Results

Competitive Layer



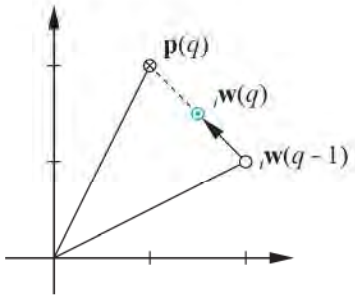
$$\mathbf{a} = \text{compet}(\mathbf{Wp})$$

Competitive Learning with the Kohonen Rule

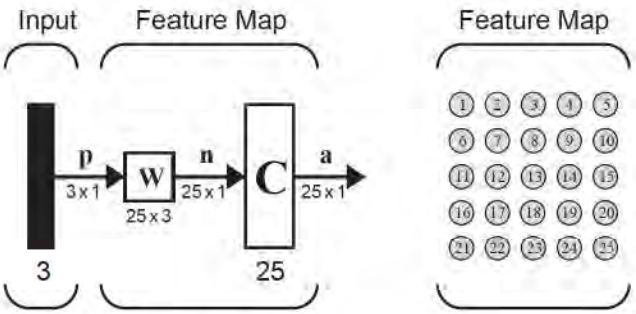
$$_{i^*}\mathbf{w}(q) = {}_{i^*}\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}(q-1)) = (1-\alpha){}_{i^*}\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

$$_{i^*}\mathbf{w}(q) = {}_{i^*}\mathbf{w}(q-1) \qquad i \neq i^*,$$

where i^* is the winning neuron.



Self-Organizing Feature Map



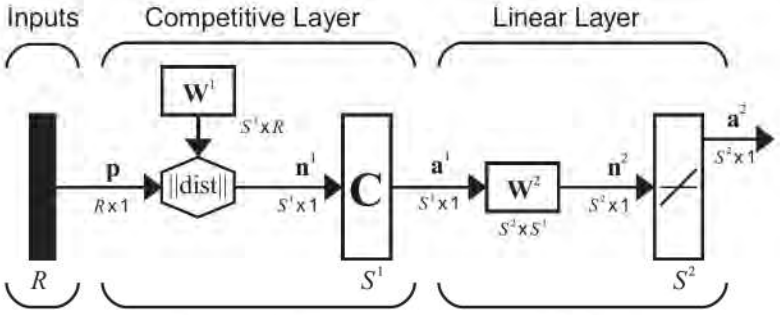
$$\mathbf{a} = \text{compet}(\mathbf{Wp})$$

Self-Organizing with the Kohonen Rule

$$
\begin{aligned}
{}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\
&= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)
\end{aligned}
\qquad i \in N_{i^*}(d)
$$

$$N_i(d) = \{j, d_{ij} \le d\}$$

LVQ Network



$$n^1_i = \|_i\mathbf{w}^1 - \mathbf{p}\| \qquad \mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1$$

$$\mathbf{a}^1 = \text{compet}(\mathbf{n}^1)$$

$$(w^2_{ki} = 1) \Rightarrow \text{subclass } i \text{ is a part of class } k$$

LVQ Network Learning with the Kohonen Rule

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a^2_{k^*} = t_{k^*} = 1$$

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a^2_{k^*} = 1 \ne t_{k^*} = 0$$

Solved Problems

P16.1 Figure P16.1 shows several clusters of normalized vectors.

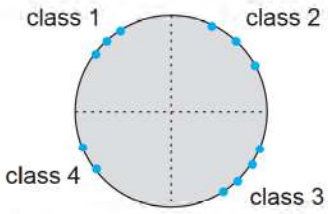


Figure P16.1 Clusters of Input Vectors for Problem P16.1

Design the weights of the competitive network shown in Figure P16.2, so that it classifies the vectors according to the classes indicated in the diagram and with the minimum number of neurons.

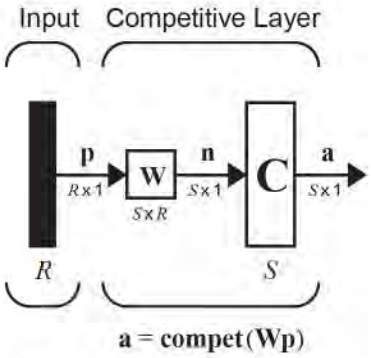


Figure P16.2 Competitive Network for Problem P16.1

Redraw the diagram showing the weights you chose and the decision boundaries that separate the region of each class.

Since there are four classes to be defined, the competitive layer will need four neurons. The weights of each neuron act as prototypes for the class that neuron represents. Therefore, for each neuron we will choose a prototype vector that appears to be approximately at the center of a cluster.

Classes 1, 2 and 3 each appear to be roughly centered at a multiple of 45° . Given this, the following three vectors are normalized (as is required for the competitive layer) and point in the proper directions.

$$ {}_1\mathbf{w} = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \; {}_2\mathbf{w} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \; {}_3\mathbf{w} = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} $$

The center of the fourth cluster appears to be about twice as far from the vertical axis as it is from the horizontal axis. The resulting normalized weight vector is

$$ {}_4\mathbf{w} = \begin{bmatrix} -2/\sqrt{5} \\ -1/\sqrt{5} \end{bmatrix}. $$

The weight matrix $\mathbf{W}$ for the competitive layer is simply the matrix of the transposed prototype vectors:

$$ W = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ {}_3\mathbf{w}^T \\ {}_4\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \\ -2/\sqrt{5} & -1/\sqrt{5} \end{bmatrix}. $$

We get Figure P16.3 by drawing these weight vectors with arrows and bisecting the circle between each adjacent weight vector to get the class regions.

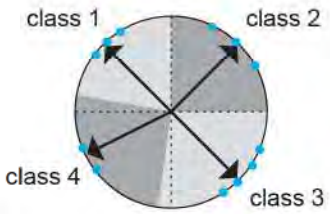


Figure P16.3 Final Classifications for Problem P16.1

P16.2 Figure P16.4 shows three input vectors and three initial weight vectors for a three-neuron competitive layer. Here are the values of the input vectors:

$$ \mathbf{p}_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}. $$

The initial values of the three weight vectors are

$$ {}_1\mathbf{w} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, {}_2\mathbf{w} = \begin{bmatrix} -2/\sqrt{5} \\ 1/\sqrt{5} \end{bmatrix}, {}_3\mathbf{w} = \begin{bmatrix} -1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix}. $$

Calculate the resulting weights found after training the competitive layer with the Kohonen rule and a learning rate α of 0.5, on the following series of inputs:

$$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3.$$

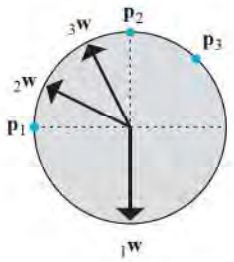


Figure P16.4 Input Vectors and Initial Weights for Problem P16.2

First we combine the weight vectors into the weight matrix $\mathbf{W}$.

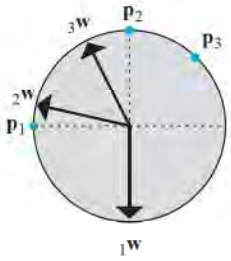
$$\mathbf{W} = \begin{bmatrix} 0 & -1 \\ -2/\sqrt{5} & 1/\sqrt{5} \\ -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}$$

Then we present the first vector $\mathbf{p}_1$.

$$\mathbf{a} = \mathbf{compet}(\mathbf{W}\mathbf{p}_1) = \mathbf{compet}\left(\begin{bmatrix} 0 & -1 \\ -2/\sqrt{5} & 1/\sqrt{5} \\ -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}\begin{bmatrix} -1 \\ 0 \end{bmatrix}\right) = \mathbf{compet}\left(\begin{bmatrix} 0 \\ 0.894 \\ 0.447 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The second neuron responded, since ${}_2\mathbf{w}$ was closest to $\mathbf{p}_1$. Therefore, we will update ${}_2\mathbf{w}$ with the Kohonen rule.

$${}_2\mathbf{w}^{new} = {}_2\mathbf{w}^{old} + \alpha(\mathbf{p}_1 - {}_2\mathbf{w}^{old}) = \begin{bmatrix} -2/\sqrt{5} \\ 1/\sqrt{5} \end{bmatrix} + 0.5\left(\begin{bmatrix} -1 \\ 0 \end{bmatrix} - \begin{bmatrix} -2/\sqrt{5} \\ 1/\sqrt{5} \end{bmatrix}\right) = \begin{bmatrix} -0.947 \\ 0.224 \end{bmatrix}$$



The diagram at left shows that the new ${}_2\mathbf{w}$ moved closer to $\mathbf{p}_1$.

We will now repeat this process for $\mathbf{p}_2$.

$$\mathbf{a} = \mathbf{compet}(\mathbf{W}\mathbf{p}_2) = \mathbf{compet}\left(\begin{bmatrix} 0 & -1 \\ -0.947 & 0.224 \\ -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = \mathbf{compet}\left(\begin{bmatrix} -1 \\ 0.224 \\ 0.894 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The third neuron won, so its weights move closer to $\mathbf{p}_2$.

$${}_3\mathbf{w}^{new} = {}_3\mathbf{w}^{old} + \alpha(\mathbf{p}_2 - {}_3\mathbf{w}^{old}) = \begin{bmatrix} -1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix} + 0.5\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} -1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix}\right) = \begin{bmatrix} -0.224 \\ 0.947 \end{bmatrix}$$

We now present $\mathbf{p}_3$.

$$\mathbf{a} = \mathbf{compet}(\mathbf{W}\mathbf{p}_3) = \mathbf{compet}\left(\begin{bmatrix} 0 & -1 \\ -0.947 & 0.224 \\ -0.224 & 0.947 \end{bmatrix}\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}\right)$$

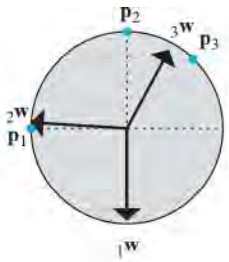
$$= \mathbf{compet}\left(\begin{bmatrix} -0.707 \\ -0.512 \\ 0.512 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The third neuron wins again.

$${}_3\mathbf{w}^{new} = {}_3\mathbf{w}^{old} + \alpha(\mathbf{p}_2 - {}_3\mathbf{w}^{old}) = \begin{bmatrix} -0.224 \\ 0.947 \end{bmatrix} + 0.5\left(\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} - \begin{bmatrix} -0.224 \\ 0.947 \end{bmatrix}\right) = \begin{bmatrix} 0.2417 \\ 0.8272 \end{bmatrix}$$

After presenting $\mathbf{p}_1$ through $\mathbf{p}_3$ again, neuron 2 will again win once and neuron 3 twice. The final weights are

$$\mathbf{W} = \begin{bmatrix} 0 & -1 \\ -0.974 & 0.118 \\ 0.414 & 0.8103 \end{bmatrix}.$$



The final weights are also shown in the diagram at left.

Note that ${}_2\mathbf{w}$ has almost learned $\mathbf{p}_1$, and ${}_3\mathbf{w}$ is directly between $\mathbf{p}_2$ and $\mathbf{p}_3$. The other weight vector, ${}_1\mathbf{w}$, was never updated. The first neuron, which never won the competition, is a dead neuron.

P16.3 Consider the configuration of input vectors and initial weights shown in Figure P16.5. Train a competitive network to cluster

these vectors using the Kohonen rule with learning rate $\alpha = 0.5$. Find graphically the position of the weights after all of the input vectors (in the order shown) have been presented once.

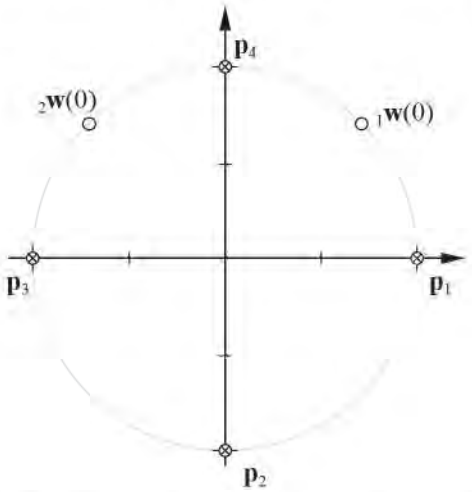


Figure P16.5 Input Vectors and Initial Weights for Problem P16.3

This problem can be solved graphically, without any computations. The results are displayed in Figure P16.6.

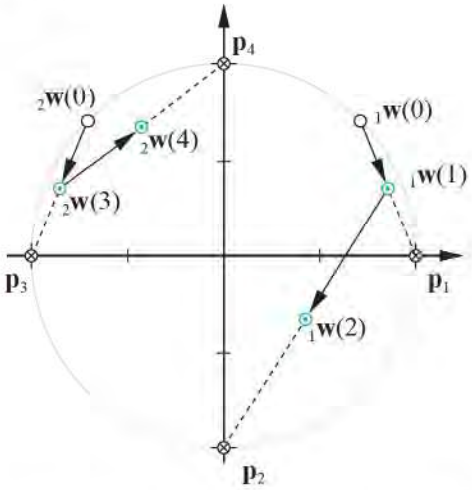


Figure P16.6 Solution for Problem P16.3

The input vector $\mathbf{p}_1$ is presented first. The weight vector ${}_1\mathbf{w}$ is closest to $\mathbf{p}_1$, therefore neuron 1 wins the competition and ${}_1\mathbf{w}$ is moved halfway to

$\mathbf{p}_1$, since $\alpha = 0.5$. Next, $\mathbf{p}_2$ is presented, and again neuron 1 wins the competition and ${}_1\mathbf{w}$ is moved halfway to $\mathbf{p}_2$. During these first two iterations, ${}_2\mathbf{w}$ is not changed.

On the third iteration, $\mathbf{p}_3$ is presented. This time ${}_2\mathbf{w}$ wins the competition and is moved halfway to $\mathbf{p}_3$. On the fourth iteration, $\mathbf{p}_4$ is presented, and neuron 2 again wins. The weight vector ${}_2\mathbf{w}$ is moved halfway to $\mathbf{p}_4$.

If we continue to train the network, neuron 1 will classify the input vectors $\mathbf{p}_1$ and $\mathbf{p}_2$, and neuron 2 will classify the input vectors $\mathbf{p}_3$ and $\mathbf{p}_4$. If the input vectors were presented in a different order, would the final classification be different?

P16.4 So far in this chapter we have only talked about feature maps whose neurons are arranged in two dimensions. The feature map shown in Figure P16.7 contains nine neurons arranged in one dimension.

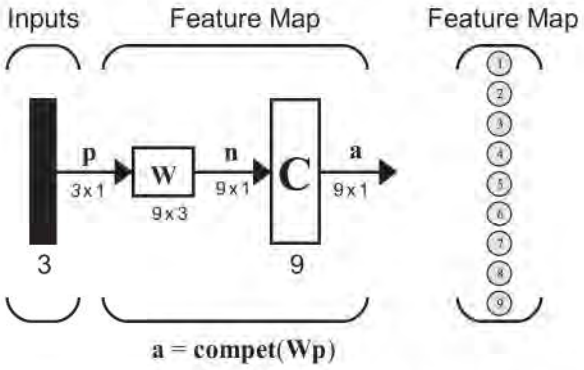


Figure P16.7 Nine-Neuron Feature Map

Given the following initial weights, draw a diagram of the weight vectors, with lines connecting weight vectors of neighboring neurons.

$$\mathbf{W} = \begin{bmatrix} 0.41 & 0.45 & 0.41 & 0 & 0 & 0 & -0.41 & -0.45 & -0.41 \\ 0.41 & 0 & -0.41 & 0.45 & 0 & -0.45 & 0.41 & 0 & -0.41 \\ 0.82 & 0.89 & 0.82 & 0.89 & 1 & 0.89 & 0.82 & 0.89 & 0.82 \end{bmatrix}^T$$

Train the feature map for one iteration, on the vector below, using a learning rate of 0.1 and a neighborhood of radius 1. Redraw the diagram for the new weight matrix.

$$\mathbf{p} = \begin{bmatrix} 0.67 \\ 0.07 \\ 0.74 \end{bmatrix}$$

The feature map diagram for the initial weights is given in Figure P16.8.

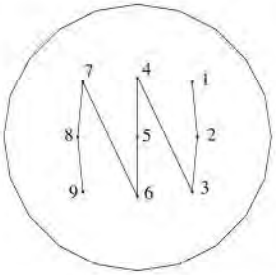


Figure P16.8 Original Feature Map

We start updating the network by presenting $\mathbf{p}$ to the network.

$$\mathbf{a} = \mathbf{compet}(\mathbf{Wp})$$

$$= \mathbf{compet}\left(\begin{bmatrix} 0.41 & 0.45 & 0.41 & 0 & 0 & 0 & -0.41 & -0.45 & -0.41 \\ 0.41 & 0 & -0.41 & 0.45 & 0 & -0.45 & 0.41 & 0 & -0.41 \\ 0.82 & 0.89 & 0.82 & 0.89 & 1 & 0.89 & 0.82 & 0.89 & 0.82 \end{bmatrix}^T \begin{bmatrix} 0.67 \\ 0.07 \\ 0.74 \end{bmatrix}\right)$$

$$= \mathbf{compet}(\begin{bmatrix} 0.91 & 0.96 & 0.85 & 0.70 & 0.74 & 0.63 & 0.36 & 0.36 & 0.3 \end{bmatrix}^T)$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

The second neuron won the competition. Looking at the network diagram, we see that the second neuron's neighbors, at a radius of 1, include neurons 1 and 3. We must update each of these neurons' weights with the Kohonen rule.

$$ {}_1\mathbf{w}(1) = {}_1\mathbf{w}(0) + \alpha(\mathbf{p} - {}_1\mathbf{w}(0)) = \begin{bmatrix} 0.41 \\ 0.41 \\ 0.82 \end{bmatrix} + 0.1\left(\begin{bmatrix} 0.67 \\ 0.07 \\ 0.74 \end{bmatrix} - \begin{bmatrix} 0.41 \\ 0.41 \\ 0.82 \end{bmatrix}\right) = \begin{bmatrix} 0.43 \\ 0.37 \\ 0.81 \end{bmatrix} $$

$$ {}_2\mathbf{w}(1) = {}_2\mathbf{w}(0) + \alpha(\mathbf{p} - {}_2\mathbf{w}(0)) = \begin{bmatrix} 0.45 \\ 0 \\ 0.89 \end{bmatrix} + 0.1\left(\begin{bmatrix} 0.67 \\ 0.07 \\ 0.74 \end{bmatrix} - \begin{bmatrix} 0.45 \\ 0 \\ 0.89 \end{bmatrix}\right) = \begin{bmatrix} 0.47 \\ 0.01 \\ 0.88 \end{bmatrix} $$

$$ {}_3\mathbf{w}(1) = {}_3\mathbf{w}(0) + \alpha(\mathbf{p} - {}_3\mathbf{w}(0)) = \begin{bmatrix} 0.41 \\ -0.41 \\ 0.82 \end{bmatrix} + 0.1\left(\begin{bmatrix} 0.67 \\ 0.07 \\ 0.74 \end{bmatrix} - \begin{bmatrix} 0.41 \\ -0.41 \\ 0.82 \end{bmatrix}\right) = \begin{bmatrix} 0.43 \\ -0.36 \\ 0.81 \end{bmatrix} $$

Figure P16.9 shows the feature map after the weights were updated.

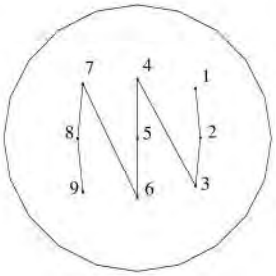


Figure P16.9 Feature Map after Update

P16.5 Given the LVQ network shown in Figure P16.10 and the weight values shown below, draw the regions of the input space that make up each class.

$$ \mathbf{W}^1 = \begin{bmatrix} 0 & 0 \\ 1 & -1 \\ 1 & 1 \\ -1 & 1 \\ -1 & -1 \end{bmatrix}, \quad \mathbf{W}^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} $$

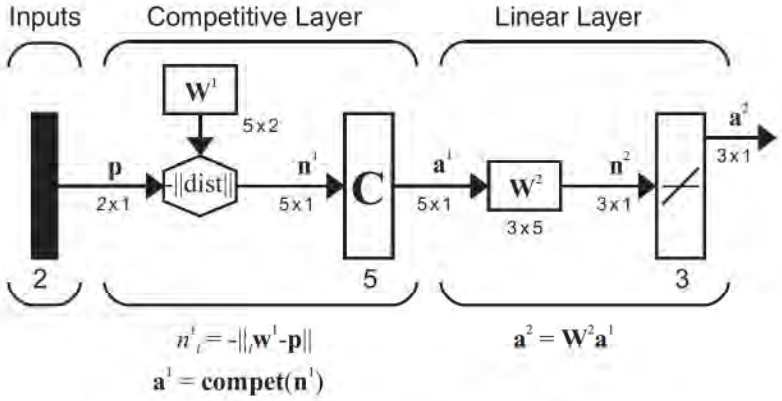


Figure P16.10 LVQ Network for Problem P16.5

We create the diagram shown in Figure P16.11 by marking each vector ${}_i\mathbf{w}$ in $\mathbf{W}^1$ according to the index k of the corresponding nonzero element in the ith column of $\mathbf{W}^2$, which indicates the class.

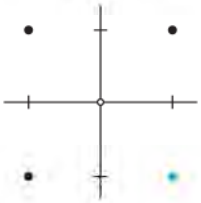


Figure P16.11 Prototype Vectors Marked by Class

The decision boundaries separating each class are found by drawing lines between each pair of prototype vectors, perpendicular to an imaginary line connecting them and equidistant from each vector.

In Figure P16.12, each convex region is colored according to the weight vector it is closest to.

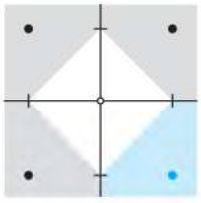


Figure P16.12 Class Regions and Decision Boundaries

P16.6 Design an LVQ network to solve the classification problem shown in Figure P16.13. The vectors in the diagram are to be classified into one of three classes, according to their color.

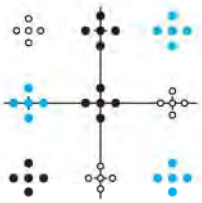


Figure P16.13 Classification Problem

When the design is complete, draw a diagram showing the region for each class.

We will begin by noting that since LVQ networks calculate the distance between vectors directly, instead of using the inner product, they can classify vectors that are not normalized, such as those above.

Next we will identify each color with a class:

- Class 1 will include all white dots.
- Class 2 will include all black dots.
- Class 3 will include all blue dots.

Now we can choose the dimensions of the LVQ network. Since there are three classes, the network must have three neurons in its output layer. There are nine subclasses (i.e., clusters). Therefore the hidden layer must have nine neurons. This gives us the network shown in Figure P16.14.

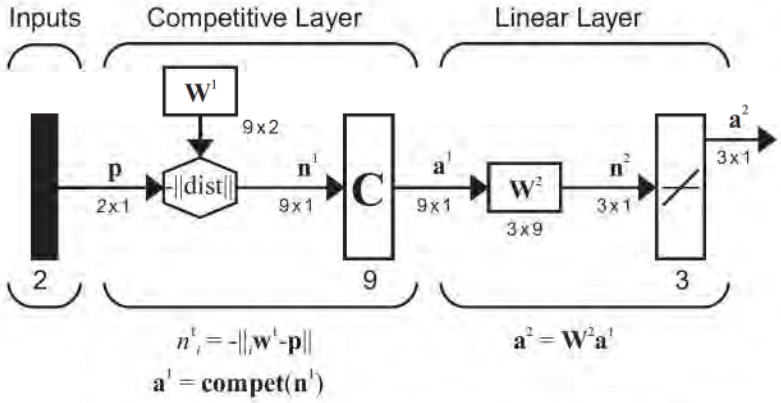


Figure P16.14 LVQ Network for Problem P16.6

We can now design the weight matrix $\mathbf{W}^1$ of the first layer by setting each row equal to a transposed prototype vector for one cluster. Picking prototype vectors at the center of each cluster gives us the following values:

$$\mathbf{W}^1 = \begin{bmatrix} -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & -1 \end{bmatrix}^T.$$

Now each neuron in the first layer will respond to a different cluster.

Next we choose $\mathbf{W}^2$ so that each subclass is connected to the appropriate class. To do this we use the following rule:

If subclass i is to be assigned to class k, then set $w_{ki}^2 = 1$.

For example, the first subclass is the top-left cluster in the vector diagram. The vectors in this cluster are white, so they belong in the first class. Therefore we should set $w_{1,1}^2$ to one.

Once we have done this for all nine subclasses we end up with these values:

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can test the network by presenting a vector to it. Here we calculate the output of the first layer for $\mathbf{p} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$:

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1) = \mathbf{compet}\left(\begin{bmatrix} -\sqrt{5} \\ -\sqrt{2} \\ -1 \\ -2 \\ -1 \\ 0 \\ -\sqrt{5} \\ -\sqrt{2} \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The network says that the vector we presented is in the sixth subclass. Let's see what the second layer says.

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 = \begin{bmatrix} 1&0&0&0&0&1&0&1&0 \\ 0&1&0&0&1&0&1&0&0 \\ 0&0&1&1&0&0&0&0&1 \end{bmatrix} \begin{bmatrix} 0\\0\\0\\0\\0\\1\\0\\0\\0 \end{bmatrix} = \begin{bmatrix} 1\\0\\0 \end{bmatrix}$$

The second layer indicates that the vector is in class 1, as indeed it is. The diagram of class regions and decision boundaries is shown in Figure P16.15.

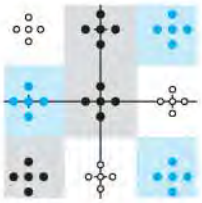


Figure P16.15 Class Regions and Decision Boundaries

P16.7 Competitive layers and feature maps require that input vectors be normalized. But what if the available data is not normalized?

One way to handle such data is simply to normalize it before giving it to the network. This has the disadvantage that the vector magnitude information, which may be important, is lost.

Another solution is to replace the inner product expression usually used to calculate net input,

$$\mathbf{a} = \text{compet}(\mathbf{Wp}),$$

with a direct calculation of distance,

$$n_i = -\|_i\mathbf{w} - \mathbf{p}\| \text{ and } \mathbf{a} = \text{compet}(\mathbf{n}),$$

as is done with the LVQ network. This works and saves the magnitude information.

However, a third solution is to append a constant of 1 to each input vector before normalizing it. Now the change in the added element will preserve the vector magnitude information.

Normalize the following vectors using this last method:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

First we add an extra element with value 1 to each vector.

$$\mathbf{p}'_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \mathbf{p}'_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \mathbf{p}'_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Then we normalize each vector.

$$\mathbf{p}''_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} / \left\| \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\| = \begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}$$

$$\mathbf{p}''_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} / \left\| \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \right\| = \begin{bmatrix} 0 \\ 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

$$\mathbf{p}''_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} / \left\| \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\| = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Now the third element of each vector contains magnitude information, since it is equal to the inverse of the magnitude of the extended vectors.

Epilogue

In this chapter we have demonstrated how the associative instar learning rule of Chapter 15 can be combined with competitive networks, similar to the Hamming network of Chapter 3, to produce powerful self-organizing networks. By combining competition with the instar rule, each of the prototype vectors that are learned by the network become representative of a particular class of input vector. Thus the competitive networks learn to divide their input space into distinct classes. Each class is represented by one of the prototype vectors (rows of the weight matrix).

Three types of networks, all developed by Tuevo Kohonen, were discussed in this chapter. The first is the standard competitive layer. Its simple operation makes it a practical network for many problems.

The self-organizing feature map is very similar to the competitive layer, but more closely models biological on-center/off-surround networks. The result is a network that not only learns to classify input vectors, but also learns the topology of the input space.

The third network, the LVQ network, uses both unsupervised and supervised learning to recognize clusters. It uses a second layer to combine multiple convex regions into classes that can have any shape. LVQ networks can even be trained to recognize classes made up of multiple unconnected regions.

Chapters 18 and 19 will build on the networks presented in this chapter. For example, Chapter 18 will carry out a more detailed examination of lateral inhibition, on-center/off-surround networks and the biology that inspired them. In Chapter 19 we discuss a modification to the standard competitive network (called adaptive resonance theory), which solves the weight stability problem that we discussed in this chapter.

Chapter 22 presents practical tips for training competitive networks, and Chapter 26 is a case study of using self organizing feature maps on a real-world clustering problem.

Further Reading

[FrSk91] J. Freeman and D. Skapura, *Neural Networks: Algorithms, Applications, and Programming Techniques*, Reading, MA: Addison-Wesley, 1991.

This text contains code fragments for network algorithms, making the details of the networks clear.

[Koho87] T. Kohonen, *Self-Organization and Associative Memory*, 2nd Ed., Berlin: Springer-Verlag, 1987.

Kohonen introduces the Kohonen rule and several networks that use it. It provides a complete analysis of linear associative models and gives many extensions and examples.

[Hech90] R. Hecht-Nielsen, *Neurocomputing*, Reading, MA: Addison-Wesley, 1990.

This book contains a section on the history and mathematics of competitive learning.

[RuMc86] D. Rumelhart, J. McClelland et al., *Parallel Distributed Processing*, vol. 1, Cambridge, MA: MIT Press, 1986.

Both volumes of this set are classics in neural network literature. The first volume contains a chapter describing competitive layers and how they learn to detect features.

Exercises

E16.1 Suppose that the weight matrix for layer 2 of the Hamming network is given by

$$\mathbf{W}^2 = \begin{bmatrix} 1 & -\frac{3}{4} & -\frac{3}{4} \\ -\frac{3}{4} & 1 & -\frac{3}{4} \\ -\frac{3}{4} & -\frac{3}{4} & 1 \end{bmatrix}.$$

This matrix violates Eq. (16.6), since

$$\varepsilon = \frac{3}{4} > \frac{1}{S-1} = \frac{1}{2}.$$

Give an example of an output from Layer 1 for which Layer 2 will fail to operate correctly.

E16.2 Consider the input vectors and initial weights shown in Figure E16.1.

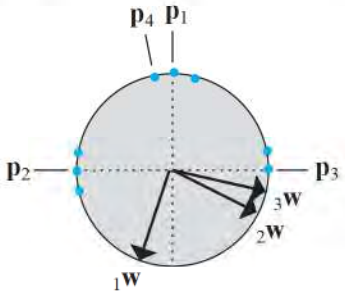


Figure E16.1 Cluster Data Vectors

i. Draw the diagram of a competitive network that could classify the data above so that each of the three clusters of vectors would have its own class.

ii. Train the network graphically (using the initial weights shown) by presenting the labeled vectors in the following order:

$$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4.$$

Recall that the competitive transfer function chooses the neuron with the lowest index to win if more than one neuron has the same

net input. The Kohonen rule is introduced graphically in Figure 16.3.

iii. Redraw the diagram in Figure E16.1, showing your final weight vectors and the decision boundaries between each region that represents a class.

E16.3 Train a competitive network using the following input patterns:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

i. Use the Kohonen learning law with $\alpha = 0.5$, and train for one pass through the input patterns. (Present each input once, in the order given.) Display the results graphically. Assume the initial weight matrix is

$$\mathbf{W} = \begin{bmatrix} \sqrt{2} & 0 \\ 0 & \sqrt{2} \end{bmatrix}.$$

ii. After one pass through the input patterns, how are the patterns clustered? (In other words, which patterns are grouped together in the same class?) Would this change if the input patterns were presented in a different order? Explain.

iii. Repeat part (i) using $\alpha = 0.25$. How does this change affect the training?

E16.4 Earlier in this chapter the term "conscience" was used to refer to a technique for avoiding the dead neuron problem plaguing competitive layers and LVQ networks.

Neurons that are too far from input vectors to ever win the competition can be given a chance by using adaptive biases that get more negative each time a neuron wins the competition. The result is that neurons that win very often start to feel "guilty" until other neurons have a chance to win.

Figure E16.2 shows a competitive network with biases. A typical learning rule for the bias b_i of neuron i is

$$b_i^{new} = \begin{cases} 0.9 b_i^{old}, \text{ if } i \neq i^* \\ b_i^{old} - 0.2, \text{ if } i = i^* \end{cases}.$$

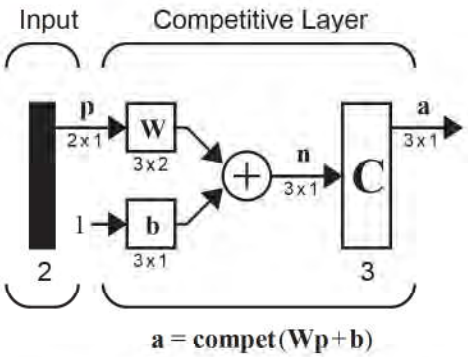


Figure E16.2 Competitive Layer with Biases

i. Examine the vectors in Figure E16.3. Is there any order in which the vectors can be presented that will cause ${}_1w$ to win the competition and move closer to one of the vectors? (Note: assume that adaptive biases are *not* being used.)

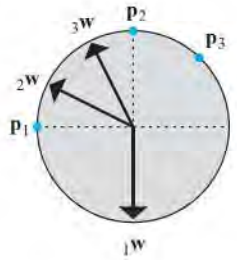


Figure E16.3 Input Vectors and Dead Neuron

ii. Given the input vectors and the initial weights and biases defined below, calculate the weights (using the Kohonen rule) and the biases (using the above bias rule). Repeat the sequence shown below until neuron 1 wins the competition.

$$\mathbf{p}_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

$${}_1\mathbf{w} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, {}_2\mathbf{w} = \begin{bmatrix} -2/\sqrt{5} \\ -1/\sqrt{5} \end{bmatrix}, {}_3\mathbf{w} = \begin{bmatrix} -1/\sqrt{5} \\ -2/\sqrt{5} \end{bmatrix}, b_1(0) = b_2(0) = b_3(0) = 0$$

Sequence of input vectors: $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \ldots$

iii. How many presentations occur before ${}_1\mathbf{w}$ wins the competition?

E16.5 The net input expression for LVQ networks calculates the distance between the input and each weight vector directly, instead of using the inner product. The result is that the LVQ network does not require normalized input vectors. This technique can also be used to allow a competitive layer to classify nonnormalized vectors. Such a network is shown in Figure E16.4.

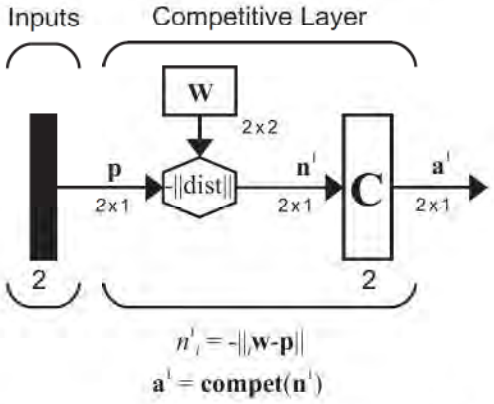


Figure E16.4 Competitive Layer with Alternate Net Input Expression

Use this technique to train a two-neuron competitive layer on the (nonnormalized) vectors below, using a learning rate, α, of 0.5.

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}$$

Present the vectors in the following order:

$$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_1.$$

Here are the initial weights of the network:

$${}_1\mathbf{w} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, {}_2\mathbf{w} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

E16.6 Repeat E16.5 for the following inputs and initial weights. Show the movements of the weights graphically for each step. If the network is trained for a large number of iterations, how will the three vectors be clustered in the final configuration?

$$\mathbf{p}_1 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$${}_1\mathbf{w} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, {}_2\mathbf{w} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

E16.7 We have a competitive learning problem, where the input vectors are

$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 2 \\ 2 \end{bmatrix},$$

and the initial weight matrix is

$$\mathbf{W} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

i. Use the Kohonen learning law to train a competitive network using a learning rate of $\alpha = 0.5$. (Present each vector once, in the order shown.) Use the modified competitive network of Figure E16.4, which uses negative distance, instead of inner product.

ii. Display the results of part i graphically, as in Figure 16.3. (Show all four iterations.)

iii. Where will the weights eventually converge (approximately)? Explain. Sketch the approximate final decision boundaries.

E16.8 Show that the modified competitive network of Figure E16.4, which computes distance directly, will produce the same results as the standard competitive network, which uses the inner product, when the input vectors are normalized.

E16.9 We would like a classifier that divides the interval of the input space defined below into five classes.

$$0 \le p_1 \le 1$$



i. Use MATLAB to randomly generate 100 values in the interval shown above with a uniform distribution.

ii. Square each number so that the distribution is no longer uniform.

iii. Write a MATLAB M-file to implement a competitive layer. Use the M-file to train a five-neuron competitive layer on the squared values

until its weights are fairly stable.

iv. How are the weight values of the competitive layer distributed? Is there some relationship between how the weights are distributed and how the squared input values are distributed?

E16.10 We would like a classifier that divides the square region defined below into sixteen classes of roughly equal size.

$$0 \le p_1 \le 1\,,\; 2 \le p_2 \le 3$$



i. Use MATLAB to randomly generate 200 vectors in the region shown above.

ii. Write a MATLAB M-file to implement a competitive layer with Kohonen learning. Calculate the net input by finding the distance between the input and weight vectors directly, as is done by the LVQ network, so the vectors do not need to be normalized. Use the M-file to train a competitive layer to classify the 200 vectors. Try different learning rates and compare performance.

iii. Write a MATLAB M-file to implement a four-neuron by four-neuron (two-dimensional) feature map. Use the feature map to classify the same vectors. Use different learning rates and neighborhood sizes, then compare performance.

E16.11 We want to train the following 1-D feature map (which uses distance instead of inner product to compute the net input):

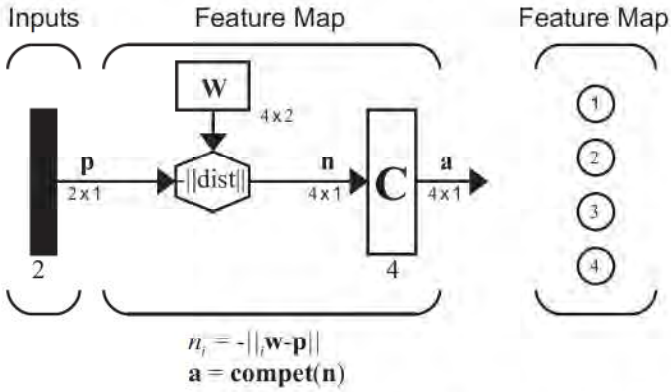


Figure E16.5 1-D Feature Map for Exercise E16.11

The initial weight matrix is $\mathbf{W}(0) = \begin{bmatrix} 2 & -1 & -1 & 1 \\ 2 & 1 & -2 & 0 \end{bmatrix}^T$.

i. Plot the initial weight vectors as dots, and connect the neighboring weight vectors as lines (as in Figure 16.10, except that this is a 1-D feature map).

ii. The following input vector is applied to the network. Perform one iteration of the feature map learning rule. (You can do this graphically.) Use a neighborhood size of 1 and a learning rate of $\alpha = 0.5$.

$$\mathbf{p}_1 = \begin{bmatrix} -2 & 0 \end{bmatrix}^T$$

iii. Plot the new weight vectors as dots, and connect the neighboring weight vectors as lines.

E16.12 Consider the following feature map, where distance is used instead of inner product to compute the net input.

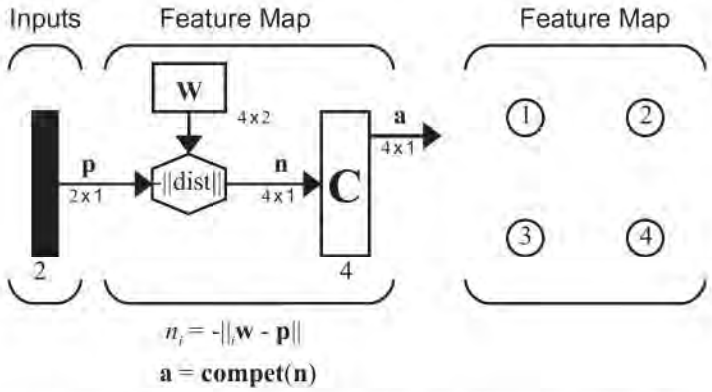


Figure E16.6 2-D Feature Map for Exercise E16.12

The initial weight matrix is

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}^T$$

i. Plot the initial weights, and show their topological connections, as in Figure 16.10.

ii. Apply the input $\mathbf{p} = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$, and perform one iteration of the feature map learning rule, with learning rate of $\alpha = 0.5$, and neighborhood radius of 1.

iii. Plot the weights after the first iteration, and show their topological connections.

E16.13 An LVQ network has the following weights:

$$\mathbf{W}^1 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

i. How many classes does this LVQ network have? How many subclasses?

ii. Draw a diagram showing the first-layer weight vectors and the decision boundaries that separate the input space into subclasses.

iii. Label each subclass region to indicate which class it belongs to.

E16.14 We would like an LVQ network that classifies the following vectors according to the classes indicated:

$$\text{class 1: } \left\{ \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right\}, \text{class 2: } \left\{ \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \right\}, \text{class 3: } \left\{ \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

i. How many neurons are required in each layer of the LVQ network?

ii. Define the weights for the first layer.

iii. Define the weights for the second layer.

iv. Test your network for at least one vector from each class.

E16.15 We would like an LVQ network that classifies the following vectors according to the classes indicated:

$$\text{class 1: } \left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \right\}, \text{class 2: } \left\{ \mathbf{p}_3 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\}$$

i. Could this classification problem be solved by a perceptron? Explain your answer.

ii. How many neurons must be in each layer of an LVQ network that can classify the above data, given that each class is made up of two convex-shaped subclasses?

iii. Define the second-layer weights for such a network.

iv. Initialize the first-layer weights of the network to all zeros and calculate the changes made to the weights by the Kohonen rule (with a learning rate α of 0.5) for the following series of vectors:

$$\mathbf{p}_4, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_1, \mathbf{p}_2.$$

v. Draw a diagram showing the input vectors, the final weight vectors and the decision boundaries between the two classes.

E16.16 An LVQ network has the following weights and training data.

$$\mathbf{W}^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$\left\{\mathbf{p}_1 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix} 2 \\ -2 \end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right\},$$

$$\left\{\mathbf{p}_4 = \begin{bmatrix} -2 \\ 0 \end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\}$$

i. Plot the training data input vectors and weight vectors (as in Figure 16.14).

ii. Perform four iterations of the LVQ learning rule, with learning rate $\alpha = 0.5$, as you present the following sequence of input vectors: $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_4$ (one iteration for each input). Do this graphically, on a separate diagram from part i.

iii. After completing the iterations in part ii, on a new diagram, sketch the regions of the input space that make up each subclass and each class. Label each region to indicate which class it belongs to.

E16.17 An LVQ network has the following weights:

$$\mathbf{W}^1 = \begin{bmatrix} 0 & 1 & -1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 1 & -1 & -1 & 1 \end{bmatrix}^T, \mathbf{W}^2 = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

i. How many classes does this LVQ network have? How many subclasses?

ii. Draw a diagram showing the first-layer weight vectors and the decision boundaries that separate the input space into subclasses.

iii. Label each subclass region to indicate which class it belongs to.

iv. Suppose that an input $\mathbf{p} = \begin{bmatrix} 1 & 0.5 \end{bmatrix}^T$ from Class 1 is presented to the network. Perform one iteration of the LVQ algorithm, with $\alpha = 0.5$.

E16.18 An LVQ network has the following weights:

$$\mathbf{W}^1 = \begin{bmatrix} 0 & 0 & 2 & 1 & 1 & -1 \\ 0 & 2 & 2 & 1 & -1 & -1 \end{bmatrix}^T, \mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

i. How many classes does this LVQ network have? How many subclasses?

ii. Draw a diagram showing the first-layer weight vectors and the decision boundaries that separate the input space into subclasses.

iii. Label each subclass region to indicate which class it belongs to.

iv. Perform one iteration of the LVQ algorithm, with the following input/target pair: $\mathbf{p} = \begin{bmatrix} -1 & -2 \end{bmatrix}^T$, $\mathbf{t} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$. Use learning rate $\alpha = 0.5$.

17 Radial Basis Networks

Objectives	17-1
Theory and Examples	17-2
Radial Basis Network	17-2
Function Approximation	17-4
Pattern Classification	17-6
Global vs. Local	17-9
Training RBF Networks	17-9
Linear Least Squares	17-11
Orthogonal Least Squares	17-17
Clustering	17-22
Nonlinear Optimization	17-23
Other Training Techniques	17-25
Summary of Results	17-26
Solved Problems	17-29
Epilogue	17-33
Further Reading	17-34
Exercises	17-36

Objectives

The multilayer networks discussed in Chapters 11 and 12 represent one type of neural network structure for function approximation and pattern recognition. As we saw in Chapter 11, multilayer networks with sigmoid transfer functions in the hidden layers and linear transfer functions in the output layer are universal function approximators. In this chapter we will discuss another type of universal approximation network, the radial basis function network. This network can be used for many of the same applications as multilayer networks.

This chapter will follow the structure of Chapter 11. We will begin by demonstrating, in an intuitive way, the universal approximation capabilities of the radial basis function network. Then we will describe three different techniques for training these networks. They can be trained by the same gradient-based algorithms discussed in Chapters 11 and 12, with derivatives computed using a form of backpropagation. However, they can also be trained using a two-stage process, in which the first layer weights are computed independently from the weights in the second layer. Finally, these networks can be built in an incremental way - one neuron at a time.

Theory and Examples

The radial basis function network is related to the multilayer perceptron network of Chapter 11. It is also a universal approximator and can be used for function approximation or pattern recognition. We will begin this chapter with a description of the network and a demonstration of its abilities for function approximation and pattern recognition.

The original work in radial basis functions was performed by Powell and others during the 1980's [Powe87]. In this original work, radial basis functions were used for exact interpolation in a multidimensional space. In other words, the function created by the radial basis interpolation was required to pass exactly through all targets in the training set. The use of radial basis functions for exact interpolation continues to be an important application area, and it is also an active area of research.

For our purposes, however, we will not be considering exact interpolation. Neural networks are often used on noisy data, and exact interpolation often results in overfitting when the training data is noisy, as we discussed in Chapter 13. Our interest is in the use of radial basis functions to provide robust approximations to unknown functions based on generally limited and noisy measurements. Broomhead and Lowe [BrLo88] were the first to develop the radial basis function neural network model, which produces a smooth interpolating function. No attempt is made to force the network response to exactly match target outputs. The emphasis is on producing networks that will generalize well to new situations.

In the next section we will demonstrate the capabilities of the radial basis function neural network. In the following sections we will describe procedures for training these networks.

Radial Basis Network

RBF

The radial basis network is a two-layer network. There are two major distinctions between the radial basis function (RBF) network and a two layer perceptron network. First, in layer 1 of the RBF network, instead of performing an inner product operation between the weights and the input (matrix multiplication), we calculate the distance between the input vector and the rows of the weight matrix. (This is similar to the LVQ network shown in Figure 16.13.) Second, instead of adding the bias, we multiply by the bias. Therefore, the net input for neuron i in the first layer is calculated as follows:

$$n_i^1 = \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\| b_i^1 . \tag{17.1}$$

Each row of the weight matrix acts as a center point - a point where the net input value will be zero. The bias performs a scaling operation on the transfer (basis) function, causing it to stretch or compress.

We should note that most papers and texts on RBF networks use the terms standard deviation, variance or spread constant, rather than bias. We have used the bias in order to maintain a consistency with other networks in this text. This is simply a matter of notation and pedagogy. The operation of the network is not affected. When a Gaussian transfer function is used, the bias is related to the standard deviation as follows: $b = 1/(\sigma\sqrt{2})$.

The transfer functions used in the first layer of the RBF network are different than the sigmoid functions generally used in the hidden layers of multilayer perceptrons (MLP). There are several different types of transfer function that can be used (see [BrLo88]), but for clarity of presentation we will consider only the Gaussian function, which is the one most commonly used in the neural network community. It is defined as follows

$$a = e^{-n^2}, \tag{17.2}$$

and it is plotted in Figure 17.1.

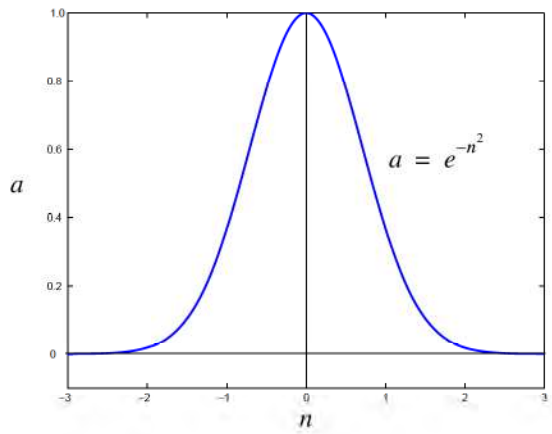


Figure 17.1 Gaussian Basis Function

Local Function
Global Function

A key property of this function is that it is *local*. This means that the output is close to zero if you move very far in either direction from the center point. This is in contrast to the *global* sigmoid functions, whose output remains close to 1 as the net input goes to infinity.

The second layer of the RBF network is a standard linear layer:

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2 \tag{17.3}$$

Figure 17.2 shows the complete RBF network.

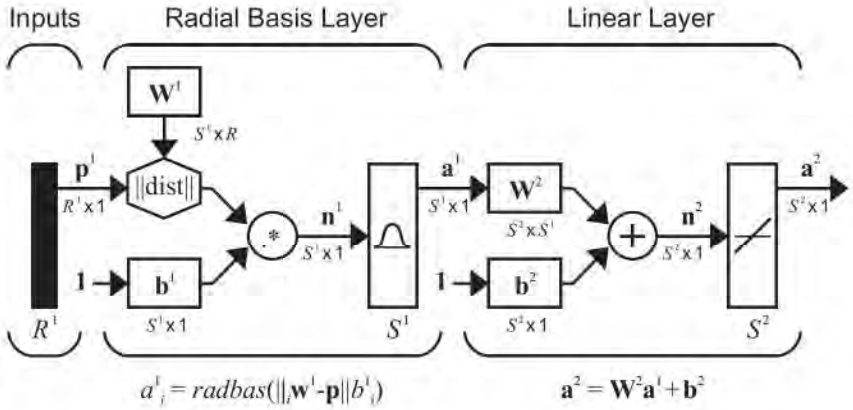


Figure 17.2 Radial Basis Network

Function Approximation

This RBF network has been shown to be a universal approximator [PaSa93], just like the MLP network. To illustrate the capability of this network, consider a network with two neurons in the hidden layer, one output neuron, and with the following default parameters:

$$w^1_{1,1} = -1,\ w^1_{2,1} = 1,\ b^1_1 = 2,\ b^1_2 = 2,$$

$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = 0.$$

The response of the network with the default parameters is shown in Figure 17.3, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$.

Notice that the response consists of two hills, one for each of the Gaussian neurons (basis functions) in the first layer. By adjusting the network parameters, we can change the shape and location of each hill, as we will see in the following discussion. (As you proceed through this example, it may be helpful to compare the response of this sample RBF network with the response of the sample MLP network in Figure 11.5.)

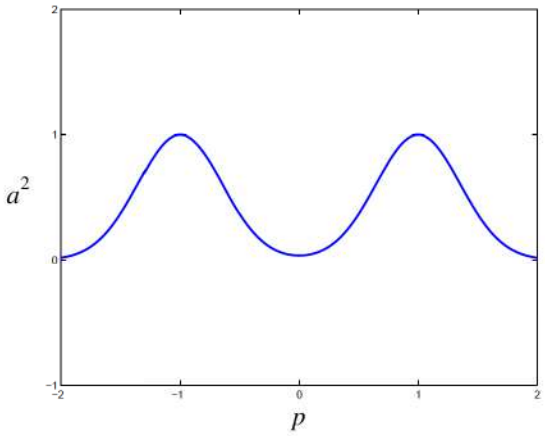


Figure 17.3 Default Network Response

Figure 17.4 illustrates the effects of parameter changes on the network response. The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$0 \le w^1_{2,1} \le 2\,,\ -1 \le w^2_{1,1} \le 1\,,\ 0.5 \le b^1_2 \le 8\,,\ -1 \le b^2 \le 1\,. \qquad (17.4)$$

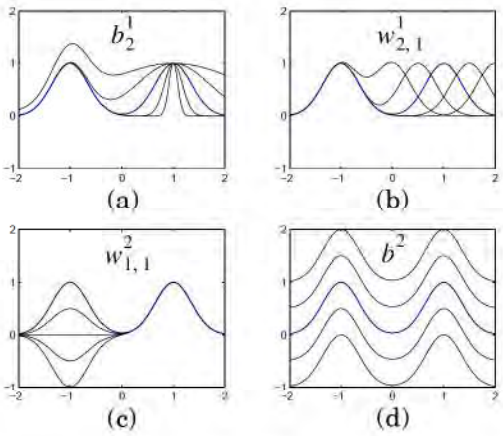


Figure 17.4 Effect of Parameter Changes on Network Response

Figure 17.4 (a) shows how the network biases in the first layer can be used to change the width of the hills - the larger the bias, the narrower the hill. Figure 17.4 (b) illustrates how the weights in the first layer determine the

Center

location of the hills; there will be a hill centered at each first layer weight. For multidimensional inputs there will be a hill centered at each row of the weight matrix. For this reason, each row of the first layer weight matrix is often called the *center* for the corresponding neuron (basis function).

Notice that the effects of the weight and the bias in first layer of the RBF network are much different than for the MLP network, which was shown in Figure 11.6. In the MLP network, the sigmoid functions create steps. The weights change the slopes of the steps, and the biases change the locations of the steps.

Figure 17.4 (c) illustrates how the weights in the second layer scale the height of the hills. The bias in the second layer shifts the entire network response up or down, as can be seen in Figure 17.4 (d). The second layer of the RBF network is the same type of linear layer used in the MLP network of Figure 11.6, and it performs a similar function, which is to create a weighted sum of the outputs of the layer 1 neurons.

This example demonstrates the flexibility of the RBF network for function approximation. As with the MLP, it seems clear that if we have enough neurons in the first layer of the RBF network, we can approximate virtually any function of interest, and [PaSa93] provides a mathematical proof that this is the case. However, although both MLP and RBF networks are universal approximators, they perform their approximations in different ways. For the RBF network, each transfer function is only active over a small region of the input space - the response is *local*. If the input moves far from a given center, the output of the corresponding neuron will be close to zero. This has consequences for the design of RBF networks. We must have centers adequately distributed throughout the range of the network inputs, and we must select biases in such a way that all of the basis functions overlap in a significant way. (Recall that the biases change the width of each basis function.) We will discuss these design considerations in more detail in later sections.



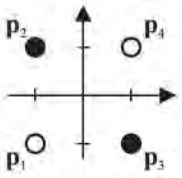
To experiment with the response of this RBF network, use the MATLAB® Neural Network Design Demonstration RBF Network Function (`nnd17nf`).

Pattern Classification



To illustrate the capabilities of the RBF network for pattern classification, consider again the classic exclusive-or (XOR) problem. The categories for the XOR gate are

$$\text{Category 1: } \left\{\mathbf{p}_2 = \begin{bmatrix}-1\\1\end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}\right\}, \text{ Category 2: } \left\{\mathbf{p}_1 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}\right\}.$$



The problem is illustrated graphically in the figure to the left. Because the two categories are not linearly separable, a single-layer network cannot perform the classification.

RBF networks can classify these patterns. In fact, there are many different RBF solutions. We will consider one solution that demonstrates in a simple way how to use RBF networks for pattern classification. The idea will be to have the network produce outputs greater than zero when the input is near patterns $\mathbf{p}_2$ or $\mathbf{p}_3$, and outputs less than zero for all other inputs. (Note that the procedures we will use to design this example network are not suitable for complex problems, but they will help us illustrate the capabilities of the RBF network.)

From the problem statement, we know that the network will need to have two inputs and one output. For simplicity, we will use only two neurons in the first layer (two basis functions), since this will be sufficient to solve the XOR problem. As we discussed earlier, the rows of the first-layer weight matrix will create centers for the two basis functions. We will choose the centers to be equal to the patterns $\mathbf{p}_2$ and $\mathbf{p}_3$. By centering a basis function at each pattern, we can produce maximum network outputs there. The first layer weight matrix is then

$$\mathbf{W}^1 = \begin{bmatrix} \mathbf{p}_2^T \\ \mathbf{p}_3^T \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{17.5}$$

The choice of the bias in the first layer depends on the width that we want for each basis function. For this problem, we would like the network function to have two distinct peaks at $\mathbf{p}_2$ and $\mathbf{p}_3$. Therefore, we don't want the basis functions to overlap too much. The centers of the basis functions are each a distance of $\sqrt{2}$ from the origin. We want the basis function to drop significantly from its peak in this distance. If we use a bias of 1, we would get the following reduction in that distance:

$$a = e^{-n^2} = e^{-(1 \cdot \sqrt{2})^2} = e^{-2} = 0.1353. \tag{17.6}$$

Therefore, each basis function will have a peak of 1 at the centers, and will drop to 0.1353 at the origin. This will work for our problem, so we select the first layer bias vector to be

$$\mathbf{b}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{17.7}$$

The original basis function response ranges from 0 to 1 (see Figure 17.1). We want the output to be negative for inputs much different than $\mathbf{p}_2$ and $\mathbf{p}_3$, so we will use a bias of -1 for the second layer, and we will use a value

of 2 for the second layer weights, in order to bring the peaks back up to 1. The second layer weights and biases then become

$$\mathbf{W}^2 = \begin{bmatrix} 2 & 2 \end{bmatrix}, \; b^2 = \begin{bmatrix} -1 \end{bmatrix}. \tag{17.8}$$

For the network parameter values given in (17.5), (17.7) and (17.8), the network response is shown in Figure 17.5. This figure also shows where the surface intersects the plane at $a^2 = 0$, which is where the decision takes place. This is also indicated by the contours shown underneath the surface. These are the function contours where $a^2 = 0$. They are almost circles that surround the $\mathbf{p}_2$ and $\mathbf{p}_3$ vectors. This means that the network output will be greater than 0 only when the input vector is near the $\mathbf{p}_2$ and $\mathbf{p}_3$ vectors.

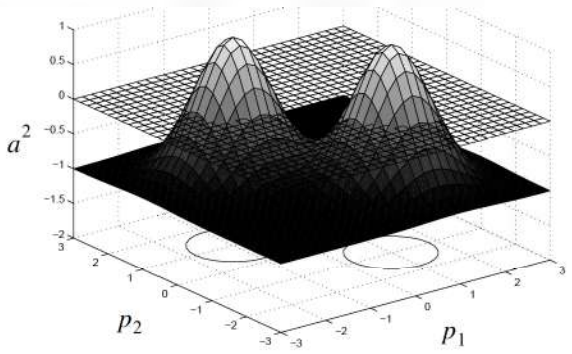


Figure 17.5 Example 2-Input RBF Function Surface

Figure 17.6 illustrates more clearly the decision boundaries. Whenever the input falls in the blue regions, the output of the network will be greater than zero. Whenever the network input falls outside the blue regions, the network output will be less than zero.

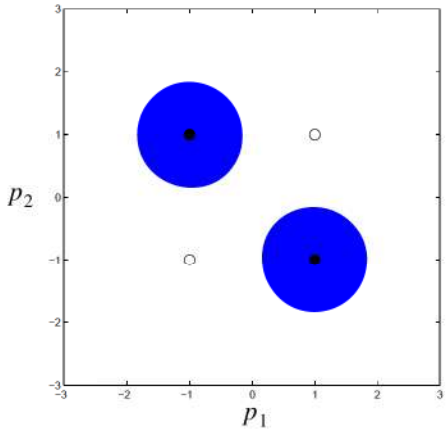


Figure 17.6 RBF Example Decision Regions

This network, therefore, classifies the patterns correctly. It is not the best solution, in the sense that it does not always assign input patterns to the closest prototype vector, unlike the MLP solution shown in Figure 11.2. You will notice that the decision regions for this RBF network are circles, unlike the linear boundaries that we see in single layer perceptrons. The MLP can put linear boundaries together to create arbitrary decision regions. The RBF network can put circular boundaries together to create arbitrary decision regions. In this problem, the linear boundaries are more efficient. Of course, when many neurons are used, and the centers are close together, the elementary RBF boundaries are no longer purely circular, and the elementary MLP boundaries are no longer purely linear. However, associating circular boundaries with RBF networks and linear boundaries with MLP networks can be helpful in understanding their operation as pattern classifiers.



To experiment with the RBF network for pattern classification, use the MATLAB® Neural Network Design Demonstration RBF Pattern Classification (nnd17pc).

Now that we see the power of RBF networks for function approximation and pattern recognition, the next step is to develop general training algorithms for these networks.

Global vs. Local

Before we discuss the training algorithms, we should say a final word about the advantages and disadvantages of the global transfer functions used by the MLP networks and the local transfer functions used by the RBF networks. The MLP creates a distributed representation, because all of the transfer functions overlap in their activity. At any given input value, many

sigmoid functions in the first layer will have significant outputs. They must sum or cancel in the second layer in order to produce the appropriate response at each point. In the RBF network, each basis function is only active over a small range of the input. For any given input, only a few basis functions will be active.

There are advantages and disadvantages to each approach. The global approach tends to require fewer neurons in the hidden layer, since each neuron contributes to the response over a large part of the input space. For the RBF network, however, basis centers must be spread throughout the range of the input space in order to provide an accurate approximation. This leads to the problem of the "curse of dimensionality," which we will discuss in the next section. Also, if more neurons, and therefore more parameters, are used, then there is a greater likelihood that the network will overfit the training data and fail to generalize well to new situations.

On the other hand, the local approach generally leads to faster training, especially when the two-stage algorithms, which will be discussed in the next section, are used. Also, the local approach can be very useful for adaptive training, in which the network continues to be incrementally trained while it is being used, as in adaptive filters (nonlinear versions of the filters in Chapter 10) or controllers. If, for a period of time, training data only appears in a certain region of the input space, then a global representation will tend to improve its accuracy in those regions at the expense of its representation in other regions. Local representations will not have this problem to the same extent. Because each neuron is only active in a small region of the input space, its weights will not be adjusted if the input falls outside that region.

Training RBF Networks

Unlike the MLP network, which is almost always trained by some gradient-based algorithm (steepest descent, conjugate gradient, Levenberg-Marquardt, etc.), the RBF network can be trained by a variety of approaches.

RBF networks can be trained using gradient-based algorithms. However, because of the local nature of the transfer function and the way in which the first layer weights and biases operate, there tend to be many more unsatisfactory local minima in the error surfaces of RBF networks than in those of MLP networks. For this reason, gradient-based algorithms are often unsatisfactory for the complete training of RBF networks. They are used on occasion, however, for fine-tuning of the network after it has been initially trained using some other method. Later in this chapter we will discuss the modifications to the backpropagation equations in Chapter 11 that are needed to compute the gradients for RBF networks.

The most commonly used RBF training algorithms have two stages, which treat the two layers of the RBF network separately. The algorithms differ

mainly in how the first layer weights and biases are selected. Once the first layer weights and biases have been selected, the second layer weights can be computed in one step, using a linear least-squares algorithm. We will discuss linear least squares in the next section.

The simplest of the two-stage algorithms arranges the centers (first layer weights) in a grid pattern throughout the input range and then chooses a constant bias so that the basis functions have some degree of overlap. This procedure is not optimal, because the most efficient approximation would place more basis functions in regions of the input space where the function to be approximated is most complex. Also, for many practical cases the full range of the input space is not used, and therefore many basis functions could be wasted. One of the drawbacks of the RBF network, especially when the centers are selected on a grid, is that they suffer from the *curse of dimensionality*. This means that as the dimension of the input space increases, the number of basis functions required increases geometrically. For example, suppose that we have one input variable, and we specify a grid of 10 basis functions evenly spaced across the range of that input variable. Now, increase the number of input variables to 2. To maintain the same grid coverage for both input variables, we would need 10^2, or 100 basis functions.

Curse of Dimensionality

Another method for selecting the centers is to select some random subset of the input vectors in the training set. This ensures that basis centers will be placed in areas where they will be useful to the network. However, due to the randomness of the selection, this procedure is not optimal. A more efficient approach is to use a method such as the Kohonen competitive layer or the feature map, described in Chapter 16, to cluster the input space. The cluster centers then become basis function centers. This ensures that the basis functions are placed in regions with significant activity. We will discuss this method in a later section.

A final procedure that we will discuss for RBF training is called orthogonal least squares. It is based on a general method for building linear models called subset selection. This method starts with a large number of possible centers—typically all of the input vectors from the training data. At each stage of the procedure, it selects one center to add to the first layer weight. The selection is based on how much the new neuron will reduce the sum squared error. Neurons are added until some criteria is met. The criteria is typically chosen to maximize the generalization capability of the network.

Linear Least Squares

In this section we will assume that the first layer weights and biases of the RBF network are fixed. This can be done by fixing the centers on a grid, or by randomly selecting the centers from the input vectors in the training data set (or by using the clustering method which is described in a later section). When the centers are randomly selected, all of the biases can be computed using the following formula [Lowe89]:

$$b_i^1 = \frac{\sqrt{S^1}}{d_{max}}, \tag{17.9}$$

where d_{max} is the maximum distance between neighboring centers. This is designed to ensure an appropriate degree of overlap between the basis functions. Using this method, all of the biases have the same value. There are other methods which use different values for each bias. We will discuss one such method later, in the section on clustering.

Once the first layer parameters have been set, the training of the second layer weights and biases is equivalent to training a linear network, as in Chapter 10. For example, consider that we have the following training points

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{17.10}$$

where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. The output of the first layer for each input $\mathbf{p}_q$ in the training set can be computed as

$$n_{i,q}^1 = \left\| \mathbf{p}_q - {}_i\mathbf{w}^1 \right\| b_i^1, \tag{17.11}$$

$$\mathbf{a}_q^1 = \mathbf{radbas}(\mathbf{n}_q^1). \tag{17.12}$$

Since the first layer weights and biases will not be adjusted, the training data set for the second layer then becomes

$$\{\mathbf{a}_1^1, \mathbf{t}_1\}, \{\mathbf{a}_2^1, \mathbf{t}_2\}, \ldots, \{\mathbf{a}_Q^1, \mathbf{t}_Q\}. \tag{17.13}$$

The second layer response is linear:

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2. \tag{17.14}$$

We want to select the weights and biases in this layer to minimize the sum square error performance index over the training set:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q^2)^T (\mathbf{t}_q - \mathbf{a}_q^2) \tag{17.15}$$

Our derivation of the solution to this linear least squares problem will follow the linear network derivation starting with Eq. (10.6). To simplify the discussion, we will assume a scalar target, and we will lump all of the parameters we are adjusting, including the bias, into one vector:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w}^2 \\ b^2 \end{bmatrix}. \tag{17.16}$$

Similarly, we include the bias input "1" as a component of the input vector

$$\mathbf{z}_q = \begin{bmatrix} \mathbf{a}_q^1 \\ 1 \end{bmatrix}. \tag{17.17}$$

Now the network output, which we usually write in the form

$$a_q^2 = ({}_1\mathbf{w}^2)^T \mathbf{a}_q^1 + b^2, \tag{17.18}$$

can be written as

$$a_q = \mathbf{x}^T \mathbf{z}_q. \tag{17.19}$$

This allows us to conveniently write out an expression for the sum square error:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (e_q)^2 = \sum_{q=1}^{Q} (t_q - a_q)^2 = \sum_{q=1}^{Q} (t_q - \mathbf{x}^T \mathbf{z}_q)^2. \tag{17.20}$$

To express this in matrix form, we define the following matrices:

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_Q \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} {}_1\mathbf{u}^T \\ {}_2\mathbf{u}^T \\ \vdots \\ {}_Q\mathbf{u}^T \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_Q^T \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_Q \end{bmatrix}. \tag{17.21}$$

The error can now be written

$$\mathbf{e} = \mathbf{t} - \mathbf{U}\mathbf{x}, \tag{17.22}$$

and the performance index become

$$F(\mathbf{x}) = (\mathbf{t} - \mathbf{U}\mathbf{x})^T (\mathbf{t} - \mathbf{U}\mathbf{x}). \tag{17.23}$$

If we use regularization, as we discussed in Chapter 13, to help in preventing overfitting, we obtain the following form for the performance index:

$$F(\mathbf{x}) = (\mathbf{t}-\mathbf{U}\mathbf{x})^T(\mathbf{t}-\mathbf{U}\mathbf{x}) + \rho\sum_{i=1}^{n} x_i^2 = (\mathbf{t}-\mathbf{U}\mathbf{x})^T(\mathbf{t}-\mathbf{U}\mathbf{x}) + \rho\mathbf{x}^T\mathbf{x}, \quad (17.24)$$

where $\rho = \alpha/\beta$ from Eq. (13.4). Let's expand this expression to obtain

$$\begin{aligned} F(\mathbf{x}) &= (\mathbf{t}-\mathbf{U}\mathbf{x})^T(\mathbf{t}-\mathbf{U}\mathbf{x}) + \rho\mathbf{x}^T\mathbf{x} = \mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{U}\mathbf{x} + \mathbf{x}^T\mathbf{U}^T\mathbf{U}\mathbf{x} + \rho\mathbf{x}^T\mathbf{x} \\ &= \mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{U}\mathbf{x} + \mathbf{x}^T[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x} \end{aligned} \quad (17.25)$$

Take a close look at Eq. (17.25), and compare it with the general form of the quadratic function, given in Eq. (8.35) and repeated here:

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}. \quad (17.26)$$

Our performance function is a quadratic function, where

$$c = \mathbf{t}^T\mathbf{t}, \ \mathbf{d} = -2\mathbf{U}^T\mathbf{t} \text{ and } \mathbf{A} = 2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]. \quad (17.27)$$

From Chapter 8 we know that the characteristics of the quadratic function depend primarily on the Hessian matrix $\mathbf{A}$. For example, if the eigenvalues of the Hessian are all positive, then the function will have one unique global minimum.

In this case the Hessian matrix is $2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]$, and it can be shown that this matrix is either positive definite or positive semidefinite (see Exercise E17.4), which means that it can never have negative eigenvalues. We are left with two possibilities. If the Hessian matrix has only positive eigenvalues, the performance index will have one unique global minimum (see Figure 8.7). If the Hessian matrix has some zero eigenvalues, the performance index will either have a weak minimum (see Figure 8.9) or no minimum (see Problem P8.7), depending on the vector $\mathbf{d}$. In this case, it must have a minimum, since $F(\mathbf{x})$ is a sum square function, which cannot be negative.

Now let's locate the stationary point of the performance index. From our previous discussion of quadratic functions in Chapter 8, we know that the gradient is

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{U}^T\mathbf{t} + 2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x}. \quad (17.28)$$

The stationary point of $F(\mathbf{x})$ can be found by setting the gradient equal to zero:

$$-2\mathbf{Z}^T\mathbf{t} + 2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x} = 0 \quad \Rightarrow \quad [\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x} = \mathbf{U}^T\mathbf{t}. \quad (17.29)$$

Therefore, the optimum weights $\mathbf{x}^*$ can be computed from

$$[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x}^* = \mathbf{U}^T\mathbf{t}. \tag{17.30}$$

If the Hessian matrix is positive definite, there will be a unique stationary point, which will be a strong minimum:

$$\mathbf{x}^* = [\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]^{-1}\mathbf{U}^T\mathbf{t} \tag{17.31}$$

Let's demonstrate this procedure with a simple problem.

Example



To illustrate the least squares algorithm, let's choose a network and apply it to a particular problem. We will use an RBF network with three neurons in the first layer to approximate the following function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2. \tag{17.32}$$

To obtain our training set we will evaluate this function at six values of p:

$$p = \{-2, -1.2, -0.4, 0.4, 1.2, 2\}. \tag{17.33}$$

This produces the targets

$$t = \{0, 0.19, 0.69, 1.3, 1.8, 2\}. \tag{17.34}$$

We will choose the basis function centers to be spaced equally throughout the input range: -2, 0 and 2. For simplicity, we will choose the bias to be the reciprocal of the spacing between points. This produces the following first layer weight and bias.

$$\mathbf{W}^1 = \begin{bmatrix} -2 \\ 0 \\ 2 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}. \tag{17.35}$$

The next step is to compute the output of the first layer, using the following equations.

$$n_{i,q}^1 = \left\| \mathbf{p}_q - {}_i\mathbf{w}^1 \right\| b_i^1, \tag{17.36}$$

$$\mathbf{a}_q^1 = \mathbf{radbas}(\mathbf{n}_q^1). \tag{17.37}$$

This produces the following $\mathbf{a}^1$ vectors

$$\mathbf{a}^1 = \left\{ \begin{bmatrix} 1 \\ 0.368 \\ 0.018 \end{bmatrix}, \begin{bmatrix} 0.852 \\ 0.698 \\ 0.077 \end{bmatrix}, \begin{bmatrix} 0.527 \\ 0.961 \\ 0.237 \end{bmatrix}, \begin{bmatrix} 0.237 \\ 0.961 \\ 0.527 \end{bmatrix}, \begin{bmatrix} 0.077 \\ 0.698 \\ 0.852 \end{bmatrix}, \begin{bmatrix} 0.018 \\ 0.368 \\ 1 \end{bmatrix} \right\} \tag{17.38}$$

We can use Eq. (17.17) and Eq. (17.21) to create the **U** and **t** matrices

$$\mathbf{U}^T = \begin{bmatrix} 1 & 0.852 & 0.527 & 0.237 & 0.077 & 0.018 \\ 0.368 & 0.698 & 0.961 & 0.961 & 0.698 & 0.368 \\ 0.018 & 0.077 & 0.237 & 0.527 & 0.852 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \tag{17.39}$$

$$\mathbf{t}^T = \begin{bmatrix} 0 & 0.19 & 0.69 & 1.3 & 1.8 & 2 \end{bmatrix}. \tag{17.40}$$

The next step is to solve for the weights and biases in the second layer using Eq. (17.30). We will begin with the regularization parameter set to zero.

$$\begin{aligned} \mathbf{x}^* &= [\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]^{-1}\mathbf{U}^T\mathbf{t} \\ &= \begin{bmatrix} 2.07 & 1.76 & 0.42 & 2.71 \\ 1.76 & 3.09 & 1.76 & 4.05 \\ 0.42 & 1.76 & 2.07 & 2.71 \\ 2.71 & 4.05 & 2.71 & 6 \end{bmatrix}^{-1} \begin{bmatrix} 1.01 \\ 4.05 \\ 4.41 \\ 6 \end{bmatrix} = \begin{bmatrix} -1.03 \\ 0 \\ 1.03 \\ 1 \end{bmatrix} \end{aligned} \tag{17.41}$$

The second layer weight and bias are therefore

$$\mathbf{W}^2 = \begin{bmatrix} -1.03 & 0 & 1.03 \end{bmatrix}, \ \mathbf{b}^2 = \begin{bmatrix} 1 \end{bmatrix}. \tag{17.42}$$

Figure 17.7 illustrates the operation of this RBF network. The blue line represents the RBF approximation, and the circles represent the six data points. The dotted lines in the upper axis represent the individual basis functions scaled by the corresponding weights in the second layer (including the constant bias term). The sum of the dotted lines will produce the blue line. In the small axis at the bottom, you can see the unscaled basis functions, which are the outputs of the first layer.

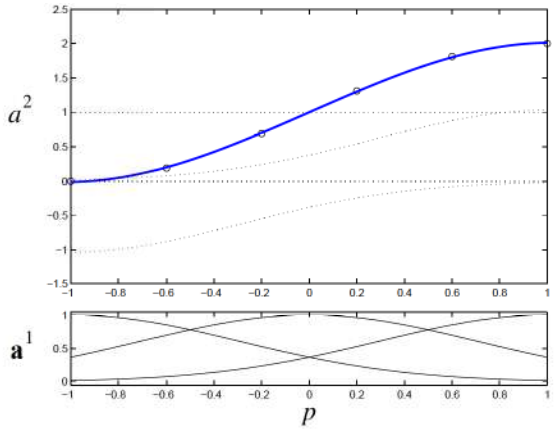


Figure 17.7 RBF Sine Approximation

The RBF network design process can be sensitive to the choice of the center locations and the bias. For example, if we select six basis functions and six data points, and if we choose the first layer biases to be 8, instead of 0.5, then the network response will be as shown in Figure 17.8. The spread of the basis function decreases as the inverse of the bias. When the bias is this large, there is not sufficient overlap in the basis functions to provide a smooth approximation. We match each data point exactly. However, because of the local nature of the basis function, the approximation to the true function is not accurate between the training data points.

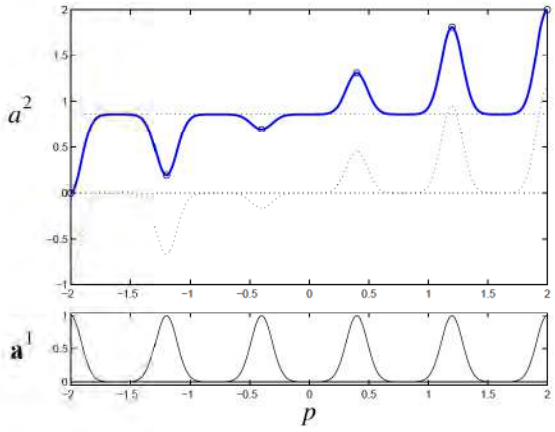


Figure 17.8 RBF Response with Bias Too Large



To experiment with the linear least squares fitting, use the MATLAB® Neural Network Design Demonstration RBF Linear Least Squares (`nnd17lls`).

Orthogonal Least Squares

In the previous section we assumed that the weights and biases in the first layer were fixed. (For example, the centers could be fixed on a grid, or randomly selected from the input vectors in the training set.) In this section we consider a different approach for selecting the centers. We will assume that there exists a number of potential centers. These centers could include the entire set of input vectors in the training set, vectors chosen in a grid pattern, or vectors chosen by any other procedure one might think of. We will then select vectors one at a time from this set of potential centers, until the network performance is satisfactory. We will build up the network one neuron at a time.

Subset Selection

The basic idea behind this method comes from statistics, and it is called *subset selection* [Mill90]. The general objective of subset selection is to choose an appropriate subset of independent variables to provide the most efficient prediction of a target dependent variable. For example, suppose that we have 10 independent variables, and we want to use them to predict our target dependent variable. We want to create the simplest predictor possible, so we want to use the minimum number of independent variables for the prediction. Which subset of the 10 independent variables should we use? The optimal approach, called an exhaustive search, tries all combinations of subsets and finds the smallest one that provides satisfactory performance. (We will define later what we mean by satisfactory performance.)

Unfortunately, this strategy is not practical. If we have Q variables in our original set, the following expression gives the number of distinct subsets:

$$\sum_{q=1}^{Q} \frac{Q!}{q!(Q-q)!}. \tag{17.43}$$

If $Q = 10$, this number is 1023. If $Q = 20$, the number is more than 1 million. We need to have a less expensive strategy than the exhaustive search. There are several suboptimal procedures. They are not guaranteed to find the optimal subset, but they require significantly less computation. One

Forward Selection

procedure is called *forward selection*. This method begins with an empty model and then adds variables one at a time. At each stage, we add the independent variable that provides the largest reduction in squared error. We stop adding variables when the performance is adequate. Another ap-

Backward Elimination

proach, called *backward elimination*, starts with all independent variables selected for the model. At each stage we eliminate the variable that would cause the least increase in the squared error. The process continues until the performance is inadequate. There are other approaches which combine

forward selection and backward elimination, so that variables can be added and deleted at each iteration.

Any of the standard subset selection techniques can be used for selecting RBF centers. For purposes of illustration, we will consider one specific form of forward selection, called orthogonal least squares [ChCo91]. Its main feature is that it efficiently calculates the error reduction provided by the addition of each potential center to the RBF network.

To develop the orthogonal least squares algorithm, we begin with Eq. (17.22), repeated here in slightly different form:

$$\mathbf{t} = \mathbf{U}\mathbf{x} + \mathbf{c}. \tag{17.44}$$

We will use our standard notation for matrix rows and columns to individually identify both the rows and the columns of the matrix $\mathbf{U}$:

$$\mathbf{U} = \begin{bmatrix} {}_1\mathbf{u}^T \\ {}_2\mathbf{u}^T \\ \vdots \\ {}_Q\mathbf{u}^T \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_Q^T \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \end{bmatrix} \tag{17.45}$$

Here each row of the matrix $\mathbf{U}$ represents the output of layer 1 of the RBF network for one input vector from the training set. There will be a column of the matrix $\mathbf{U}$ for each neuron (basis function) in layer 1 plus the bias term ($n = S^1 + 1$). Note that for the OLS algorithm, the potential centers for the basis functions are often chosen to be all of the input vectors in the training set. In this case, n will equal $Q + 1$, since the constant "1" for the bias term is included in $\mathbf{z}$, as shown in Eq. (17.17).

Regression Matrix

Eq. (17.44) is in the form of a standard linear regression model. The matrix $\mathbf{U}$ is called the *regression matrix*, and the columns of $\mathbf{U}$ are called the regressor vectors.

The objective of OLS is to determine how many columns of $\mathbf{U}$ (numbers of neurons or basis functions) should be used. The first step is to calculate how much each potential column would reduce the squared error. The problem is that the columns are generally correlated with each other, and so it is difficult to determine how much each individual column would reduce the error. For this reason, we need to first orthogonalize the columns. Orthogonalizing the columns means that we can decompose $\mathbf{U}$ as follows:

$$\mathbf{U} = \mathbf{M}\mathbf{R}, \tag{17.46}$$

where $\mathbf{R}$ is an upper triangular matrix, with ones on the diagonal:

$$\mathbf{R} = \begin{bmatrix} 1 & r_{1,2} & r_{1,3} & \cdots & r_{1,n} \\ 0 & 1 & r_{2,3} & \cdots & r_{2,n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & r_{n-1,n} \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{17.47}$$

and $\mathbf{M}$ is a matrix with orthogonal columns $\mathbf{m}_i$. This means that $\mathbf{M}$ has the following properties:

$$\mathbf{M}^T\mathbf{M} = \mathbf{V} = \begin{bmatrix} v_{1,1} & 0 & \cdots & 0 \\ 0 & v_{2,2} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & v_{n,n} \end{bmatrix} = \begin{bmatrix} \mathbf{m}_1^T\mathbf{m}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{m}_2^T\mathbf{m}_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \mathbf{m}_n^T\mathbf{m}_n \end{bmatrix}. \tag{17.48}$$

Now Eq. (17.44) can be written

$$\mathbf{t} = \mathbf{MRx} + \mathbf{e} = \mathbf{Mh} + \mathbf{e}, \tag{17.49}$$

where

$$\mathbf{h} = \mathbf{Rx}. \tag{17.50}$$

The least squares solution for Eq. (17.49) is

$$\mathbf{h}^* = [\mathbf{M}^T\mathbf{M}]^{-1}\mathbf{M}^T\mathbf{t} = [\mathbf{V}]^{-1}\mathbf{M}^T\mathbf{t}, \tag{17.51}$$

and because $\mathbf{V}$ is diagonal, the elements of $\mathbf{h}^*$ can be computed

$$h_i^* = \frac{\mathbf{m}_i^T\mathbf{t}}{v_{i,i}} = \frac{\mathbf{m}_i^T\mathbf{t}}{\mathbf{m}_i^T\mathbf{m}_i}. \tag{17.52}$$

From $\mathbf{h}^*$ we can compute $\mathbf{x}^*$ using Eq. (17.50). Since $\mathbf{R}$ is upper-triangular, Eq. (17.50) can be solved by back-substitution and does not require a matrix inversion.

There are a number of ways to obtain the orthogonal vectors $\mathbf{m}_i$, but we will use the Gram-Schmidt orthogonalization process of Eq. (5.20), starting with the original columns of $\mathbf{U}$.

$$\mathbf{m}_1 = \mathbf{u}_1, \tag{17.53}$$

$$\mathbf{m}_k = \mathbf{u}_k - \sum_{i=1}^{k-1} r_{i,k}\mathbf{m}_i, \tag{17.54}$$

where

$$r_{i,k} = \frac{\mathbf{m}_i^T \mathbf{u}_k}{\mathbf{m}_i^T \mathbf{m}_i}, \; i = 1, \ldots, k-1. \tag{17.55}$$

Now let's see how orthogonalizing the columns of $\mathbf{U}$ enables us to efficiently calculate the squared error contribution of each basis vector. Using Eq. (17.49), the total sum square value of the targets is given by

$$\mathbf{t}^T\mathbf{t} = [\mathbf{M}\mathbf{h} + \mathbf{e}]^T[\mathbf{M}\mathbf{h} + \mathbf{e}] = \mathbf{h}^T\mathbf{M}^T\mathbf{M}\mathbf{h} + \mathbf{e}^T\mathbf{M}\mathbf{h} + \mathbf{h}^T\mathbf{M}^T\mathbf{e} + \mathbf{e}^T\mathbf{e}. \tag{17.56}$$

Consider the second term in the sum:

$$\mathbf{e}^T\mathbf{M}\mathbf{h} = [\mathbf{t} - \mathbf{M}\mathbf{h}]^T\mathbf{M}\mathbf{h} = \mathbf{t}^T\mathbf{M}\mathbf{h} - \mathbf{h}^T\mathbf{M}^T\mathbf{M}\mathbf{h}. \tag{17.57}$$

If we use the optimal $\mathbf{h}^*$ from Eq. (17.51), we find

$$\mathbf{e}^T\mathbf{M}\mathbf{h}^* = \mathbf{t}^T\mathbf{M}\mathbf{h}^* - \mathbf{t}^T\mathbf{M}\mathbf{V}^{-1}\mathbf{M}^T\mathbf{M}\mathbf{h}^* = \mathbf{t}^T\mathbf{M}\mathbf{h}^* - \mathbf{t}^T\mathbf{M}\mathbf{h}^* = 0. \tag{17.58}$$

Therefore the total sum square value from Eq. (17.56) becomes

$$\mathbf{t}^T\mathbf{t} = \mathbf{h}^T\mathbf{M}^T\mathbf{M}\mathbf{h} + \mathbf{e}^T\mathbf{e} = \mathbf{h}^T\mathbf{V}\mathbf{h} + \mathbf{e}^T\mathbf{e} = \sum_{i=1}^{n} h_i^2\mathbf{m}_i^T\mathbf{m}_i + \mathbf{e}^T\mathbf{e}. \tag{17.59}$$

The first term on the right of Eq. (17.59) is the contribution to the sum squared value explained by the regressors, and the second term is the remaining sum squared value that is not explained by the regressors. Therefore, regressor (basis function) i contributes

$$h_i^2\mathbf{m}_i^T\mathbf{m}_i \tag{17.60}$$

to the squared value. This also represents how much the squared error can be reduced by including the corresponding basis function in the network. We will use this number, after normalizing by the total squared value, to determine the next basis function to include at each iteration:

$$o_i = \frac{h_i^2\mathbf{m}_i^T\mathbf{m}_i}{\mathbf{t}^T\mathbf{t}}. \tag{17.61}$$

This number always falls between zero and one.

Now let's put all these ideas together into an algorithm for selecting centers.

The OLS Algorithm

To begin the algorithm, we start with all potential basis functions included in the regression matrix $\mathbf{U}$. (As we explained below Eq. (17.45), if all input vectors in the training set are to be considered potential basis function centers, then the $\mathbf{U}$ matrix will be Q by $Q+1$.) This matrix represents only potential basis functions, since we start with no basis functions included in the network.

The first stage of the OLS algorithm consists of the following three steps, for $i = 1, \ldots, Q$:

$$\mathbf{m}_1^{(i)} = \mathbf{u}_i, \tag{17.62}$$

$$h_1^{(i)} = \frac{\mathbf{m}_1^{(i)T}\mathbf{t}}{\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}, \tag{17.63}$$

$$o_1^{(i)} = \frac{(h_1^{(i)})^2\,\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}{\mathbf{t}^T\mathbf{t}}. \tag{17.64}$$

We then select the basis function that creates the largest reduction in error:

$$o_1 = o_1^{(i_1)} = max\{o_1^{(i)}\}, \tag{17.65}$$

$$\mathbf{m}_1 = \mathbf{m}_1^{(i_1)} = \mathbf{u}_{i_1}. \tag{17.66}$$

The remaining iterations of the algorithm continue as follows (for iteration k):

For $i = 1, \ldots, Q$, $i \neq i_1, \ldots, i \neq i_{k-1}$

$$r_{j,k}^{(i)} = \frac{\mathbf{m}_j^T\mathbf{u}_i}{\mathbf{m}_j^T\mathbf{m}_j}, \; j = 1, \ldots, k-1, \tag{17.67}$$

$$\mathbf{m}_k^{(i)} = \mathbf{u}_i - \sum_{j=1}^{k-1} r_{j,k}^{(i)}\mathbf{m}_j, \tag{17.68}$$

$$h_k^{(i)} = \frac{\mathbf{m}_k^{(i)T}\mathbf{t}}{\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}}, \tag{17.69}$$

$$o_k^{(i)} = \frac{(h_k^{(i)})^2 \mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}}{\mathbf{t}^T\mathbf{t}}, \tag{17.70}$$

$$o_k = o_k^{(i_k)} = max\{o_k^{(i)}\}, \tag{17.71}$$

$$r_{j,k} = r_{j,k}^{(i_k)}, j = 1, \ldots, k-1. \tag{17.72}$$

$$\mathbf{m}_k = \mathbf{m}_k^{(i_k)}. \tag{17.73}$$

The iterations continue until some stopping criterion is met. One choice of stopping criterion is

$$1 - \sum_{j=1}^{k} o_j < \delta, \tag{17.74}$$

where δ is some small number. However, if δ is chosen too small, we can have overfitting, since the network will become too complex. An alternative is to use a validation set, as we discussed in the chapter on generalization. We would stop when the error on the validation set increased.

After the algorithm has converged, the original weights $\mathbf{x}$ can be computed from the transformed weights $\mathbf{h}$ by using Eq. (17.50). This produces, by back substitution,

$$x_n = h_n, x_k = h_k - \sum_{j=k+1}^{n} r_{j,k} x_j, \tag{17.75}$$

where n is the final number of weights and biases in the second layer (adjustable parameters).



To experiment with orthogonal least squares learning, use the MATLAB® Neural Network Design Demonstration RBF Orthogonal Least Squares (`nnd17ols`).

Clustering

There is another approach [MoDa89] for selecting the weights and biases in the first layer of the RBF network. This method uses the competitive networks described in Chapter 16. Recall that the competitive layer of Kohonen (see Figure 16.2) and the Self Organizing Feature Map (see Figure

16.9) perform a clustering operation on the input vectors of the training set. After training, the rows of the competitive networks contain prototypes, or cluster centers. This provides an approach for locating centers and selecting biases for the first layer of the RBF network. If we take the input vectors from the training set and perform a clustering operation on them, the resulting prototypes (cluster centers) could be used as centers for the RBF network. In addition, we could compute the variance of each individual cluster and use that number to calculate an appropriate bias to be used for the corresponding neuron.

Consider again the following training set:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}. \tag{17.76}$$

We want to perform a clustering of the input vectors from this training set:

$$\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\}. \tag{17.77}$$

We will train the first layer weights of the RBF network to perform a clustering of these vectors, using the Kohonen learning rule of Eq. (16.13), and repeated here:

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \tag{17.78}$$

where $\mathbf{p}(q)$ is one of the input vectors in the training set, and ${}_{i^*}\mathbf{w}^1(q-1)$ is the weight vector that was closest to $\mathbf{p}(q)$. (We could also use other clustering algorithms, such as the Self Organizing Feature Map, or the k-means clustering algorithm, which was suggested in [MoDa89].) As described in Chapter 16, Eq. (17.78) is repeated until the weights have converged. The resulting converged weights will represent cluster centers of the training set input vectors. This will insure that we will have basis functions located in areas where input vectors are most likely to occur.

In addition to selecting the first layer weights, the clustering process can provide us with a method for determining the first layer biases. For each neuron (basis function), locate the n_c input vectors from the training set that are closest to the corresponding weight vector (center). Then compute the average distance between the center and its neighbors.

$$dist_i = \frac{1}{n_c}\left(\sum_{j=1}^{n_c} \left\|\mathbf{p}_j^i - {}_i\mathbf{w}^1\right\|^2\right)^{\frac{1}{2}} \tag{17.79}$$

where $\mathbf{p}_1^i$ is the input vector that closest to ${}_i\mathbf{w}^1$, and is $\mathbf{p}_2^i$ the next closest input vector. From these distances, [MoDa89] recommends setting the first layer biases as follows:

$$b_i^1 = \frac{1}{\sqrt{2}dist_i}. \tag{17.80}$$

Therefore, when a cluster is wide, the corresponding basis function will be wide as well. Notice that in this case each bias in the first layer will be different. This should provide a network that is more efficient in its use of basis functions than a network with equal biases.

After the weights and biases of the first layer are determined, linear least squares is used to find the second layer weights and biases.

There is a potential drawback to the clustering method for designing the first layer of the RBF network. The method only takes into account the distribution of the input vectors; it does not consider the targets. It is possible that the function we are trying to approximate is more complex in regions for which there are fewer inputs. For this case, the clustering method will not distribute the centers appropriately. On the other hand, one would hope that the training data is located in regions where the network will be most used, and therefore the function approximation will be most accurate in those areas.

Nonlinear Optimization

It is also possible to train RBF networks in the same manner as MLP networks—using nonlinear optimization techniques, in which all weights and biases in the network are adjusted at the same time. These methods are not generally used for the full training of RBF networks, because these networks tend to have many more unsatisfactory local minima in their error surfaces. However, nonlinear optimization can be used for the fine-tuning of the network parameters, after initial training by one of the two-stage methods we presented in earlier sections.

We will not present the nonlinear optimization methods in their entirety here, since they were treated extensively in Chapters 11 and 12. Instead, we will simply indicate how the basic backpropagation algorithm for computing the gradient in MLP networks can be modified for RBF networks.

The derivation of the gradient for RBF networks follows the same pattern as the gradient development for MLP networks, starting with Eq. (11.9), which you may wish to review at this time. Here we will only discuss the one step where the two derivations differ. The difference occurs with Eq. (11.20). The net input for the second layer of the RBF network has the same form as its counterpart in the MLP network, but the first layer net input has a different form (as given in Eq. (17.1) and repeated here):

$$n_i^1 = \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\| b_i^1 = b_i^1 \sqrt{\sum_{j=1}^{S^1} (p_j - w_{i,j}^1)^2}. \tag{17.81}$$

If we take the derivative of this function with respect to the weights and biases, we obtain

$$\frac{\partial n_i^1}{\partial w_{i,j}^1} = b_i^1 \frac{1/2}{\sqrt{\sum_{j=1}^{s^1} (p_j - w_{i,j}^1)^2}} 2(p_j - w_{i,j}^1)(-1) = \frac{b_i^1 (w_{i,j}^1 - p_j)}{\left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|}, \tag{17.82}$$

$$\frac{\partial n_i^1}{\partial b_i^1} = \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|. \tag{17.83}$$

This produces the modified gradient equations (compare with Eq. (11.23) and Eq. (11.24)) for Layer 1 of the RBF network

$$\frac{\partial \hat{F}}{\partial w_{i,j}^1} = s_i^1 \frac{b_i^1 (w_{i,j}^1 - p_j)}{\left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|}, \tag{17.84}$$

$$\frac{\partial \hat{F}}{\partial b_i^1} = s_i^1 \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|. \tag{17.85}$$

Therefore, if we look at the summary of the gradient descent backpropagation algorithm for MLP networks, from Eq. (11.44) to Eq. (11.47), we find that the only difference for RBF networks is that we substitute Eq. (17.84) and Eq. (17.85) for Eq. (11.46) and Eq. (11.47) when $m = 1$. When $m = 2$ the original equations remain the same.



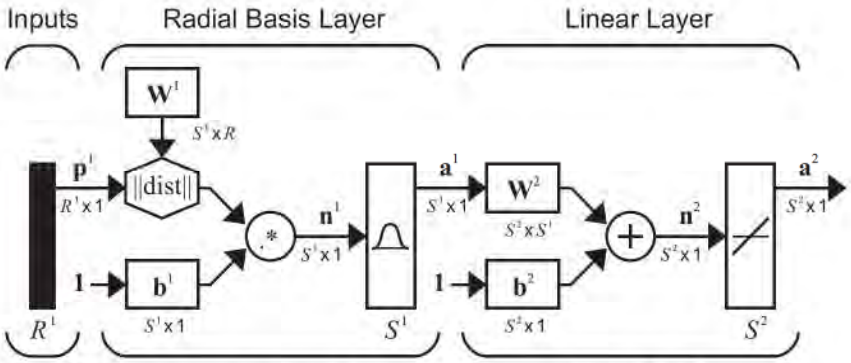
To experiment with nonlinear optimization learning, use the MATLAB® Neural Network Design Demonstration RBF Nonlinear Optimization (nnd17no).

Other Training Techniques

In this chapter we have only touched the surface of the variety of training techniques that have been proposed for RBF networks. We have attempted to present the principal concepts, but there are many variations. For example, the OLS algorithm has been extended to handle multiple outputs [ChCo92] and regularized performance indices [ChCh96]. It has also been used in combination with a genetic algorithm [ChCo99], which was used to select the first layer biases and the regularization parameter. The expectation maximization algorithm has also been suggested by several authors for optimizing the center locations, starting with [Bish91]. [OrHa00] used a regression tree approach for center selection. There have also been many variations on the use of clustering and on the combination of clustering for initialization and nonlinear optimization for fine-tuning. The architecture of the RBF network lends itself to many training approaches.

Summary of Results

Radial Basis Network



Training RBF Networks

Linear Least Squares

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w}^2 \\ b^2 \end{bmatrix}, \quad \mathbf{z}_q = \begin{bmatrix} \mathbf{a}_q^1 \\ 1 \end{bmatrix}$$

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_Q \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} {}_1\mathbf{u}^T \\ {}_2\mathbf{u}^T \\ \vdots \\ {}_Q\mathbf{u}^T \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_Q^T \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_Q \end{bmatrix}$$

$$F(\mathbf{x}) = (\mathbf{t} - \mathbf{U}\mathbf{x})^T(\mathbf{t} - \mathbf{U}\mathbf{x}) + \rho\mathbf{x}^T\mathbf{x}$$

$$[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x}^* = \mathbf{U}^T\mathbf{t}$$

Orthogonal Least Squares

Step 1

$$\mathbf{m}_1^{(i)} = \mathbf{u}_i,$$

$$h_1^{(i)} = \frac{\mathbf{m}_1^{(i)T}\mathbf{t}}{\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}},$$

$$o_1^{(i)} = \frac{(h_1^{(i)})^2\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}{\mathbf{t}^T\mathbf{t}}.$$

$$o_1 = o_1^{(i_1)} = max\{o_1^{(i)}\}$$

$$\mathbf{m}_1 = \mathbf{m}_1^{(i_1)} = \mathbf{u}_{i_1}.$$

Step k

For $i = 1, \ldots, Q,\ i \neq i_1, \ldots, i \neq i_{k-1}$

$$r_{j,k}^{(i)} = \frac{\mathbf{m}_j^T\mathbf{u}_k}{\mathbf{m}_j^T\mathbf{m}_j},\ j = 1, \ldots, k,$$

$$\mathbf{m}_k^{(i)} = \mathbf{u}_i - \sum_{j=1}^{k-1} r_{j,k}^{(i)}\mathbf{m}_j,$$

$$h_k^{(i)} = \frac{\mathbf{m}_k^{(i)T}\mathbf{t}}{\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}},$$

$$o_k^{(i)} = \frac{(h_k^{(i)})^2\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}}{\mathbf{t}^T\mathbf{t}},$$

$$o_k = o_k^{(i_k)} = max\{o_k^{(i)}\},$$

$$\mathbf{m}_k = \mathbf{m}_k^{(i_k)}.$$

Clustering

Training the weights

$$ {}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)) $$

Selecting the bias

$$ dist_i = \frac{1}{n_c}\left(\sum_{j=1}^{n_c} \left\| \mathbf{p}_j^i - {}_i\mathbf{w}^1 \right\|^2 \right)^{\frac{1}{2}} $$

$$ b_i^1 = \frac{1}{\sqrt{2}dist_i} $$

Nonlinear Optimization

Replace Eq. (11.46) and Eq. (11.47) in standard backpropagation with

$$ \frac{\partial \hat{F}}{\partial w_{i,j}^1} = s_i^1 \frac{b_i^1 (w_{i,j}^1 - p_j)}{\left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|}, $$

$$ \frac{\partial \hat{F}}{\partial b_i^1} = s_i^1 \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|. $$

Solved Problems

P17.1 Use the OLS algorithm, to approximate the following function:

$$g(p) = \cos(\pi p) \text{ for } -1 \le p \le 1 .$$

To obtain our training set we will evaluate this function at five values of p:

$$p = \{-1, -0.5, 0, 0.5, 1\} .$$

This produces the targets

$$t = \{-1, 0, 1, 0, -1\} .$$

Perform one iteration of the OLS algorithm. Assume that the inputs in the training set are the potential centers and that the biases are all equal to 1.

First, we compute the outputs of the first layer:

$$n_{i,q}^1 = \left\| \mathbf{p}_q - {}_i\mathbf{w}^1 \right\| b_i^1 ,$$

$$\mathbf{a}_q^1 = \mathbf{radbas}(\mathbf{n}_q^1) ,$$

$$\mathbf{a}^1 = \left\{ \begin{bmatrix} 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \\ 0.018 \end{bmatrix}, \begin{bmatrix} 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \end{bmatrix}, \begin{bmatrix} 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \end{bmatrix}, \begin{bmatrix} 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \end{bmatrix}, \begin{bmatrix} 0.018 \\ 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \end{bmatrix} \right\} .$$

We can use Eq. (17.17) and Eq. (17.21) to create the **U** and **t** matrices:

$$\mathbf{U}^T = \begin{bmatrix} 1.000 & 0.779 & 0.368 & 0.105 & 0.018 \\ 0.779 & 1.000 & 0.779 & 0.368 & 0.105 \\ 0.368 & 0.779 & 1.000 & 0.779 & 0.368 \\ 0.105 & 0.368 & 0.779 & 1.000 & 0.779 \\ 0.018 & 0.105 & 0.368 & 0.779 & 1.000 \\ 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \end{bmatrix} ,$$

$$\mathbf{t}^T = \begin{bmatrix} -1 & 0 & 1 & 0 & -1 \end{bmatrix} .$$

Now we perform step one of the algorithm:

$$\mathbf{m}_1^{(i)} = \mathbf{u}_i ,$$

$$\mathbf{m}_1^{(1)} = \begin{bmatrix} 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \\ 0.018 \end{bmatrix}, \mathbf{m}_1^{(2)} = \begin{bmatrix} 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \end{bmatrix}, \mathbf{m}_1^{(3)} = \begin{bmatrix} 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \end{bmatrix}, \mathbf{m}_1^{(4)} = \begin{bmatrix} 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \end{bmatrix},$$

$$\mathbf{m}_1^{(5)} = \begin{bmatrix} 0.018 \\ 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \end{bmatrix}, \mathbf{m}_1^{(6)} = \begin{bmatrix} 1.000 \\ 1.000 \\ 1.000 \\ 1.000 \\ 1.000 \end{bmatrix},$$

$$h_1^{(i)} = \frac{\mathbf{m}_1^{(i)T}\mathbf{t}}{\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}},$$

$$h_1^{(1)} = -0.371 , h_1^{(2)} = -0.045 , h_1^{(3)} = 0.106 , h_1^{(4)} = -0.045 , h_1^{(5)} = -0.371 ,$$
$$h_1^{(6)} = -0.200 ,$$

$$o_1^{(i)} = \frac{(h_1^{(i)})^2 \mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}{\mathbf{t}^T\mathbf{t}},$$

$$o_1^{(1)} = 0.0804 , o_1^{(2)} = 0.0016 , o_1^{(3)} = 0.0094 , o_1^{(4)} = 0.0016 , o_1^{(5)} = 0.0804 ,$$
$$o_1^{(6)} = 0.0667 .$$

We see that the first and fifth centers would produce a 0.0804 reduction in the error. This means that the error would be reduced by 8.04%, if the first or fifth center were used in a single-neuron first layer. We would typically select the first center, since it has the smallest index.

If we stop at this point, we would add the first center to the hidden layer. Using Eq. (17.75), we would find that $w_{1,1}^2 = x_1 = h_1 = h_1^{(1)} = -0.371$. Also, $b_2 = 0$, since the bias center, $\mathbf{m}_1^{(6)}$, was not selected on the first iteration. Note that if we continue to add neurons in the hidden layer, the first weight will change. This can be seen from Eq. (17.75). This equation to find x_k is only used after all of the h_k are found. Only x_n will exactly equal h_n .

If we continue the algorithm, the first column would be removed from $\mathbf{U}$. We would then orthogonalize all remaining columns of $\mathbf{U}$ with respect to $\mathbf{m}_1$, which was chosen on the first iteration, using Eq. (17.54). It is interesting to note that the error reduction on the second iteration would be much higher than the reduction on the first iteration. The sequence of reductions would be 0.0804, 0.3526, 0.5074, 0.0448, 0.0147, 0, and the centers would be chosen in the following order: 1, 2, 5, 3, 4, 6. The reason that reductions in later iterations are higher is that it takes a combination of basis functions to produce the best approximation. This is why forward selection is not guaranteed to produce the optimal combination, which can be found with an exhaustive search. Also, notice that the bias is selected last, and it produces no reduction in the error.

P17.2 Figure P17.1 illustrates a classification problem, where Class I vectors are represented by dark circles, and Class II vectors are represented by light circles. These categories are not linearly separable. Design a radial basis function network to correctly classify these categories.

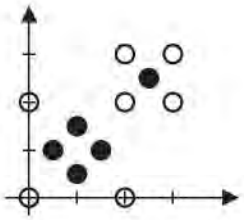


Figure P17.1 Classification Problem for Problem P17.2

From the problem statement, we know that the network will need to have two inputs, and we can use one output to distinguish the two classes. We will choose a positive output for Class I vectors, and a negative output for Class II vectors. The Class I region is made up of two simple subregions, and it appears that two neurons should be sufficient to perform the classification. The rows of the first-layer weight matrix will create centers for the two basis functions, and we will choose each center to be located in the middle of one subregion. By centering a basis function in each subregion, we can produce maximum network outputs there. The first layer weight matrix is then

$$\mathbf{W}^1 = \begin{bmatrix} 1 & 1 \\ 2.5 & 2.5 \end{bmatrix}.$$

The choice of the biases in the first layer depends on the width that we want for each basis function. For this problem, the first basis function should be wider than the second. Therefore, the first bias will be smaller than the second bias. The boundary formed by the first basis function

should have a radius of approximately 1, while the second basis function boundary should have a radius of approximately $1/2$. We want the basis functions to drop significantly from their peaks in these distances. If we use a bias of 1 for the first neuron and a bias of 2 for the second neuron, we get the following reductions within one radius of the centers:

$$a = e^{-n^2} = e^{-(1 \cdot 1)^2} = e^{-1} = 0.3679\,, \; a = e^{-n^2} = e^{-(2 \cdot 0.5)^2} = e^{-1} = 0.3679$$

This will work for our problem, so we select the first layer bias vector to be

$$\mathbf{b}^1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

The original basis function response ranges from 0 to 1 (see Figure 17.1). We want the output to be negative for inputs outside the decision regions, so we will use a bias of -1 for the second layer, and we will use a value of 2 for the second layer weights, in order to bring the peaks back up to 1. The second layer weights and biases then become

$$\mathbf{W}^2 = \begin{bmatrix} 2 & 2 \end{bmatrix}, \; b^2 = \begin{bmatrix} -1 \end{bmatrix}.$$

For these network parameter values, the network response is shown on the right side of Figure P17.2. This figure also shows where the surface intersects the plane at $a^2 = 0$, which is where the decision takes place. This is also indicated by the contours shown underneath the surface. These are the function contours where $a^2 = 0$. These decision regions are shown more clearly on the left side of Figure P17.2.

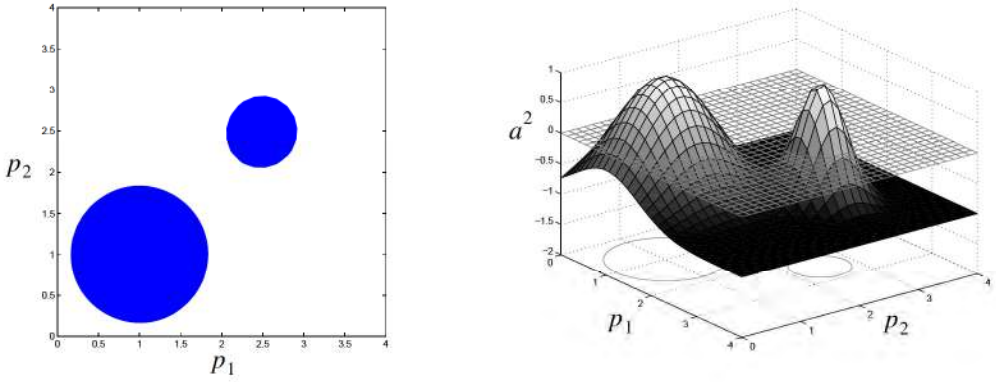


Figure P17.2 Decision Regions for Problem P17.2

P17.3 For an RBF network with one input and one neuron in the hidden layer, the initial weights and biases are chosen to be

$$w^1(0) = 0,\ b^1(0) = 1,\ w^2(0) = -2,\ b^2(0) = 1.$$

An input/target pair is given to be

$$\{p = -1, t = 1\}.$$

Perform one iteration of steepest descent backpropagation with $\alpha = 1$.

The first step is to propagate the input through the network.

$$n^1 = \|p - w^1\| b^1 = 1\sqrt{(-1-0)^2} = 1$$

$$a^1 = radbas(n^1) = e^{-(n^1)^2} = e^{-1} = 0.3679$$

$$n^2 = w^2 a^1 + b^2 = (-2)(0.3679) + 1 = 0.2642$$

$$a^2 = purelin(n^2) = n^2 = 0.2642$$

$$e = (t - a^2) = (1 - (0.2642)) = 0.7358$$

Now we backpropagate the sensitivities using Eq. (11.44) and Eq. (11.45).

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t} - \mathbf{a}) = -2[1](e) = -2[1]0.7358 = -1.4716$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = [-2n^1 e^{-(n^1)^2}]w^2\mathbf{s}^2 = [-2 \times 1 \times e^{-1}](-2)(-1.4716) = -2.1655$$

Finally, the weights and biases are updated using Eq. (11.46) and Eq. (11.47) for Layer 2, and Eq. (17.84) and Eq. (17.85) for Layer 1:

$$w^2(1) = w^2(0) - \alpha s^2 (a^1)^T = (-2) - 1(-1.4716)(0.3679) = -1.4586,$$

$$w^1(1) = w^1(0) - \alpha s^1 \left(\frac{b^1(w^1 - p)}{\|p - w^1\|}\right) = (0) - 1(-2.1655)\left(\frac{1(0-(-1))}{\|-1-0\|}\right) = 2.1655,$$

$$b^2(1) = b^2(0) - \alpha s^2 = 1 - 1(-1.4716) = 2.4716,$$

$$b^1(1) = b^1(0) - \alpha s^1 \|p - w^1\| = 1 - 1(-2.1655)\|-1-0\| = 3.1655.$$

Epilogue

The radial basis function network is an alternative to the multilayer perceptron network for problems of function approximation and pattern recognition. In this chapter we have demonstrated the operation of the RBF network, and we have described several techniques for training the network. Unlike MLP training, RBF training usually consists of two stages. In the first stage, the weights and biases in the first layer are found. In the second stage, which typically involves linear least squares, the second layer weights and biases are calculated.

Further Reading

[Bish91] C. M. Bishop, "Improving the generalization properties of radial basis function neural networks," *Neural Computation*, Vol. 3, No. 4, pp. 579-588, 1991.

First published use of the expectation-maximization algorithm for optimizing cluster centers for the radial basis network.

[BrLo88] D.S. Broomhead and D. Lowe, "Multivariable function interpolation and adaptive networks," *Complex Systems*, vol.2, pp. 321-355, 1988.

This seminal paper describes the first use of radial basis functions in the context of neural networks.

[ChCo91] S. Chen, C.F.N. Cowan, and P.M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Transactions on Neural Networks*, Vol.2, No.2, pp.302-309, 1991.

The first description of the use of the subset selection technique for selecting centers for radial basis function networks.

[ChCo92] S. Chen, P. M. Grant, and C. F. N. Cowan, "Orthogonal least squares algorithm for training multioutput radial basis function networks," *Proceedings of the Institute of Electrical Engineers*, Vol. 139, Pt. F, No. 6, pp. 378–384, 1992.

This paper extends the orthogonal least squares algorithm to the case of multiple outputs.

[ChCh96] S. Chen, E. S. Chng, and K. Alkadhimi, "Regularised orthogonal least squares algorithm for constructing radial basis function networks," *International Journal of Control*, Vol. 64, No. 5, pp. 829–837, 1996.

Modifies the orthogonal least squares algorithm to handle regularized performance indices.

[ChCo99] S. Chen, C.F.N. Cowan, and P.M. Grant, "Combined Genetic Algorithm Optimization and Regularized Orthogonal Least Squares Learning for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, Vol.10, No.5, pp.302-309, 1999.

Combines a genetic algorithm with orthogonal least squares to compute the regularization parameter and basis

function spread, while also selecting centers and solving for the optimal second layer weights of radial basis function networks.

[Lowe89] D. Lowe, "Adaptive radial basis function nonlinearities, and the problem of generalization," *Proceedings of the First IEE International Conference on Artificial Neural Networks*, pp. 171 - 175, 1989.

This paper describes the use of gradient-based algorithms for training all of the parameters of an RBF network, including basis function centers and widths. It also provides a formula for setting the basis function widths, if the centers are randomly chosen from the training data set.

[Mill90] A.J. Miller, *Subset Selection in Regression*. Chapman and Hall, N.Y., 1990.

This book provides a very complete and clear discussion of the general problem of subset selection. This involves choosing an appropriate subset from a large set of independent input variables, in order to provide the most efficient prediction of some dependent variable.

[MoDa89] J. Moody and C.J. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, Vol. 1, pp. 281–294, 1989.

The first published use of clustering methods to find radial basis function centers and variances.

[OrHa00] M. J. Orr, J. Hallam, A. Murray, and T. Leonard, "Assessing rbf networks using delve," IJNS, 2000.

This paper compares a variety of methods for training radial basis function networks. The methods include forward selection with regularization and also regression trees.

[PaSa93] J. Park and I.W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 5, pp. 305-316, 1993.

This paper proves the universal approximation capability of radial basis function networks.

[Powe87] M.J.D. Powell, "Radial basis functions for multivariable interpolation: a review," *Algorithms for Approximation*, pp. 143-167, Oxford, 1987.

This paper provides the definitive survey of the original work on radial basis functions. The original use of radial basis functions was in exact multivariable interpolation.

Exercises

E17.1 Design an RBF network to perform the classification illustrated in Figure E17.1. The network should produce a positive output whenever the input vector is in the shaded region and a negative output otherwise.

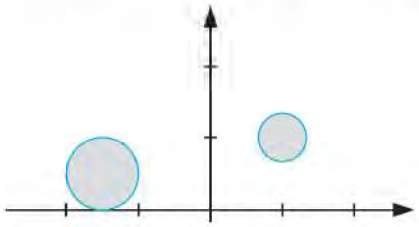


Figure E17.1 Pattern Classification Regions

E17.2 Choose the weights and biases for an RBF network with two neurons in the hidden layer and one output neuron, so that the network response passes through the points indicated by the blue circles in Figure E17.2.



Use the MATLAB® Neural Network Design Demonstration RBF Network Function (`nnd17nf`) *to check your result.*

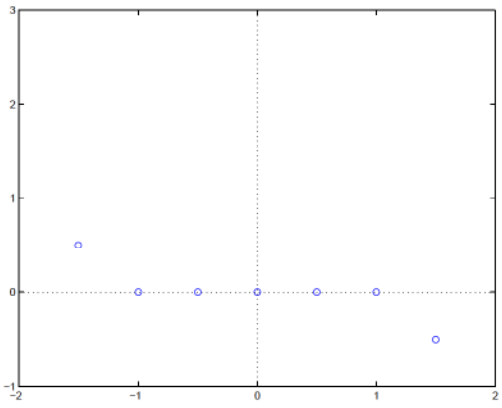


Figure E17.2 Function Approximation Exercise

E17.3 Consider a 1-2-1 RBF network (two neurons in the hidden layer and one output neuron). The first layer weights and biases are fixed as follows:

$$\mathbf{W}^1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \ \mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

Assume that the bias in the second layer is fixed at 0 ($b^2 = 0$). The training set has the following input/target pairs:

$$\{p_1 = 1, t_1 = -1\}, \ \{p_2 = 0, t_2 = 0\}, \ \{p_3 = -1, t_3 = 1\}.$$

i. Use linear least squares to solve for the second layer weights, assuming that the regularization parameter $\rho = 0$.

ii. Plot the contour plot for the sum squared error. Recall that it will be a quadratic function. (See Chapter 8.)



iii. Write a MATLAB® M-file to check your answers to parts i. and ii.

iv. Repeat parts i. to iii., with $\rho = 4$. Plot regularized squared error.

E17.4 The Hessian matrix for the performance index of the RBF network, given in Eq. (17.25), is

$$2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}].$$

Show that this matrix is at least positive semidefinite for $\rho \geq 0$, and show that it is positive definite if $\rho > 0$.

E17.5 Consider an RBF network with the weights and biases in the first layer fixed. Show how the LMS algorithm of Chapter 10 could be modified for learning the second layer weights and biases.

E17.6 Suppose that a Gaussian transfer function in the first layer of the RBF network is replaced with a linear transfer function.

i. In Solved Problem P11.8, we showed that a multilayer perceptron with linear transfer functions in each layer is equivalent to a single-layer perceptron. If we use a linear transfer function in each layer of an RBF network, is that equivalent to a single-layer network? Explain.



ii. Work out an example, equivalent to Figure 17.4, to demonstrate the operation of the RBF network with linear transfer function in the first layer. Use MATLAB® to plot your figures. Do you think that the RBF network will be a universal approximator, if the first layer transfer function is linear? Explain your answer.

E17.7 Consider a Radial Basis Network, as in Figure 17.2, but assume that there is no bias in the second layer. There are two neurons in the first layer (two basis functions). The first layer weights (centers) and biases are fixed, and we have three inputs and targets. The outputs of the first layer and the targets are given by

$$\mathbf{a}^1 = \left\{ \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}, \; t = \{0, 1, 2\}.$$

i. Use linear least squares to find the second layer weights of the network.

ii. Assume now that the basis function centers in the first layer are only potential centers. If orthogonal least squares is used to select potential centers, which center will be selected first, what will be its corresponding weight in the second layer, and how much will it reduce the squared error? Show all calculations clearly and in order.

iii. Is there a relationship between the two weights that you computed in part i. and the single weight that you computed in part ii? Explain.

E17.8 Repeat E17.7 for the following data:

i. $\mathbf{a}^1 = \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}, \; t = \{1, 2, -1\}.$

ii. $\mathbf{a}^1 = \left\{ \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}, \; t = \{3, 1, 2\}.$

E17.9 Consider the variation of the radial basis network shown in Figure E17.3. The inputs and targets in the training set are $\{p_1 = -1, t_1 = -1\}$, $\{p_2 = 1, t_2 = 1\}$.

i. Find the linear least squares solution for the weight matrix $\mathbf{W}^2$.

ii. For the weight matrix $\mathbf{W}^2$ that you found in part i., sketch the network response as the input varies from -2 to 2.

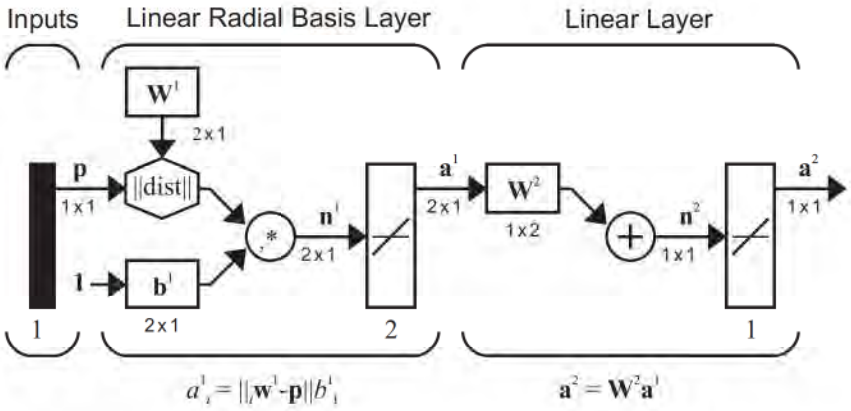


Figure E17.3 Radial Basis Network for Exercise E17.9

E17.10 Write a MATLAB® program to implement the linear least squares algorithm for the $1-S^1-1$ RBF network with first layer weights and biases fixed. Train the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{8}p\right) \text{ for } -2 \le p \le 2 .$$



i. Select 10 training points at random from the interval $-2 \le p \le 2$.

ii. Select four basis function centers evenly spaced on the interval $-2 \le p \le 2$. Then, use Eq. (17.9) to set the bias. Finally, use linear least squares to find the second layer weights and biases, assuming that there is no regularization. Plot the network response for $-2 \le p \le 2$, and show the training points on the same plot. Compute the sum squared error over the training set.

iii. Double the bias from part ii and repeat.

iv. Decrease the bias by half from part ii, and repeat.

v. Compare the final sum squared errors for all cases and explain your results.

E17.11 Use the function described in Exercise E17.10, and use an RBF network with 10 neurons in the hidden layer.

i. Repeat Exercise E17.10 ii. with regularization parameter $\rho = 0.2$. Describe the changes in the RBF network response.



ii. Add uniform random noise in the range $[-0.1, 0.1]$ to the training targets. Repeat Exercise E17.10 ii. with no regularization and with regularization parameter $\rho = 0.2, 2, 20$. Which case produces the best results. Explain.

E17.12 Write a MATLAB® program to implement the orthogonal least squares algorithm. Repeat Exercise E17.10 using the orthogonal least squares algorithm. Use the 10 random training point inputs as the potential centers, and use Eq. (17.9) to set the bias. Use only the first four selected centers. Compare your final sum squared errors with the result from E17.10 part ii.



E17.13 Write a MATLAB® program to implement the steepest descent algorithm for the $1-S^1-1$ RBF network. Train the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{8}p\right) \text{ for } -2 \le p \le 2 .$$



You should be able to use a slightly modified version of the program you wrote for Exercise E11.25.

i. Select 10 data points at random from the interval $-2 \le p \le 2$.

ii. Initialize all parameters (weights and biases in both layers) as small random numbers, and then train the network to convergence. (Experiment with the learning rate α, to determine a stable value.) Plot the network response for $-2 \le p \le 2$, and show the training points on the same plot. Compute the sum squared error over the training set. Use 2, 4 and 8 centers. Try different sets of initial weights.

iii. Repeat part ii., but use a different method for initializing the parameters. Start by setting the parameters as follows. First, select basis function centers evenly spaced on the interval $-2 \le p \le 2$. Then, use Eq. (17.9) to set the bias. Finally, use linear least squares to find the second layer weights and biases. Compute the squared error for these initial weights and biases. Starting from these initial conditions, train all parameters with steepest descent.

iv. Compare the final sum squared errors for all cases and explain your results.

E17.14 Suppose that a radial basis function layer (Layer 1 of the RBF network) were used in the second or third layer of a multilayer network. How could you modify the backpropagation equation, Eq. (11.35), to accommodate this change. (Recall that the weight update equations would be modified from Eq. (11.23) and Eq. (11.24) to Eq. (17.84) and Eq. (17.85).)

E17.15 Consider again Exercise E16.10, in which you trained a feature map to cluster the input space

$$0 \le p_1 \le 1 , \; 2 \le p_2 \le 3 .$$

Assume that over this input space, we wish to use an RBF network to approximate the following function:

$$t = \sin(2\pi p_1)\cos(2\pi p_2).$$



i. Use MATLAB to randomly generate 200 input vectors in the region shown above.

ii. Write a MATLAB M-file to implement a four-neuron by four-neuron (two-dimensional) feature map. Calculate the net input by finding the distance between the input and weight vectors directly, as is done by the LVQ network, so the vectors do not need to be normalized. Use the feature map to cluster the input vectors.

iii. Use the trained feature map weight matrix from part ii as the weight matrix of the first layer of an RBF network. Use Eq. (17.79) to determine the average distance between each cluster and its center, and then use Eq. (17.80) to set the bias for each neuron in the first layer of the RBF network.

iv. For each of the 200 input vectors in part i, compute the target response for the function above. Then use the resulting input/target pairs to determine the second-layer weights and bias for the RBF network.

v. Repeat parts ii to iv, using a two by two feature map. Compare your results.

18 Grossberg Network

Objectives	18-1
Theory and Examples	18-2
Biological Motivation: Vision	18-3
Illusions	18-4
Vision Normalization	18-8
Basic Nonlinear Model	18-9
Two-Layer Competitive Network	18-12
Layer 1	18-13
Layer 2	18-17
Choice of Transfer Function	18-20
Learning Law	18-22
Relation to Kohonen Law	18-24
Summary of Results	18-26
Solved Problems	18-30
Epilogue	18-42
Further Reading	18-43
Exercises	18-45

Objectives

This chapter is a continuation of our discussion of associative and competitive learning algorithms in Chapters 15 and 16. The Grossberg network described in this chapter is a self-organizing continuous-time competitive network. This will be the first time we have considered continuous-time recurrent networks, and we will introduce concepts here that will be further explored in Chapters 20 and 21. This Grossberg network is also the foundation for the adaptive resonance theory (ART) networks that we will present in Chapter 19.

We will begin with a discussion of the biological motivation for the Grossberg network: the human visual system. Although we will not cover this material in any depth, the Grossberg networks are so heavily influenced by biology that it is difficult to discuss his networks without putting them in their biological context. It is also important to note that biology provided the original inspirations for the field of artificial neural networks, and we should continue to look for inspiration there, as scientists continue to develop new understanding of brain function.

Theory and Examples

During the late 1960s and the 1970s the number of researchers in the field of neural networks dropped dramatically. There were, however, a number of researchers who continued to work during this period, including Tuevo Kohonen, James Anderson, Kunihiko Fukushima and Shun-ichi Amari, among others. One of the most prolific was Stephen Grossberg.

Grossberg has been continuously active, and highly productive, in neural network research since the early 1960s. His work is characterized by the use of nonlinear mathematics to model specific functions of mind and brain, and his volume of output has been consistent with the magnitude of the task of understanding the brain. The topics of his papers have ranged from such specific areas as how competitive networks can provide contrast enhancement in vision, to such general subjects as a universal theory for human memory.

In part because of the scale of his efforts, his work has a reputation for difficulty. Each new paper is built on a foundation of 30 years of previous results, and is therefore difficult to assess on its own merits. In addition, his terminology is self-consistent, but not in standard use by other researchers. His work is also characterized by a high level of mathematical and neurophysiological sophistication. He is inspired by the interdisciplinary research into brain function by Helmholtz, Maxwell and Mach, and he brings this viewpoint to his work. His research lies at the intersection of mathematics, psychology and neurophysiology. A lack of background in these areas can make his work difficult to approach on a first reading.

This chapter will take a rudimentary look at one of the seminal networks developed by Grossberg. In order to obtain the maximum understanding of his ideas, we will begin with a brief introduction to the biological motivation for the network: the visual system. Then we will present the mathematical building block for many of Grossberg's networks: the shunting model. After understanding the function of this simple model, we will demonstrate how it can be used to build a neural network for adaptive pattern recognition. This network will then form the basis for the adaptive resonance theory networks that are discussed in Chapter 19. By building up gradually to the more complex networks, we hope to make them more easily understandable.

There is an important lesson we should take from the work described in this chapter. Although the original inspiration for the field of artificial neural networks came from biology, at times we forget to look back to biology for new ideas. It will be the blending of biology, mathematics, psychology and other disciplines that will provide the maximum growth in our understanding of neural networks.

Biological Motivation: Vision

The neural network described in this chapter was inspired by the developmental physiology of the human visual system. In this section we want to provide a brief introduction to vision, so that the function of the network will be more understandable.

In Figure 18.1 we have a schematic representation of the first stages of the visual system. Light passes through the cornea (the transparent front part of the eye) and the lens, which bends the light to focus objects on the retina (the interior layer of the external wall of the eye). It is after the light falls on the retina that the immense job of translating this information into an understandable image begins. As we will see later in this chapter, much of what we "see" is not actually present in the image projected onto the retina.

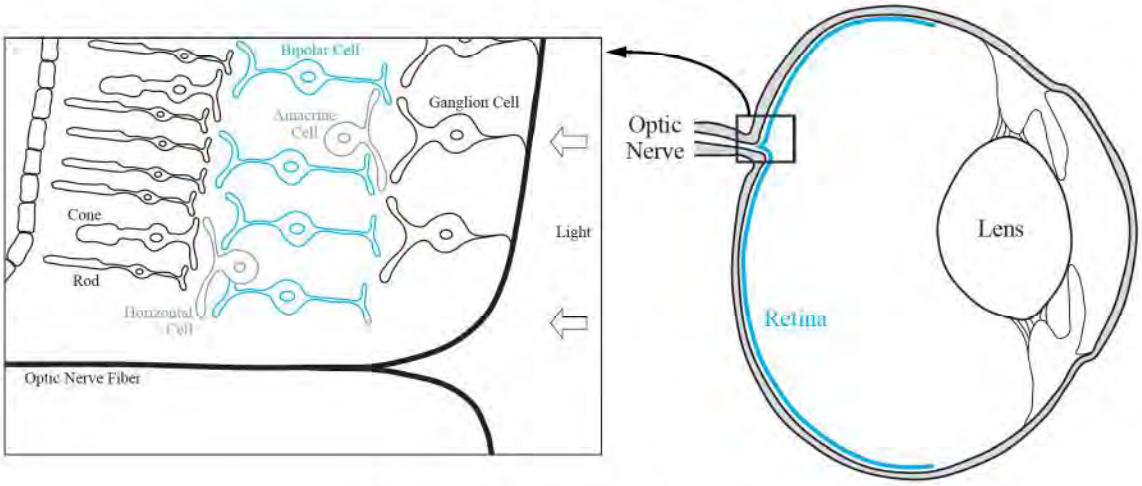


Figure 18.1 Eyeball and Retina

Retina

The *retina* is actually a part of the brain. It becomes separated from the brain during fetal development, but remains connected to it through the optic nerve. The retina consists of three layers of nerve cells. The outer layer consists of the photoreceptors (rods and cones), which convert light into electrical signals. The *rods* allow us to see in dim light, whereas the *cones* allow us to see fine detail and color. For reasons not completely understood, light must pass through the other two layers of the retina in order to stimulate the rods and cones. As we will see later, this obstruction must be compensated for in neural processing, in order to reconstruct recognizable images.

Rods
Cones

Bipolar Cells

The middle layer of the retina consists of three types of cells: bipolar cells, horizontal cells and amacrine cells. *Bipolar cells* receive input from the receptors and feed into the third layer of the retina, containing the ganglion

Horizontal Cells
Amacrine Cells

cells. *Horizontal cells* link the receptors and the bipolar cells, and *amacrine cells* link bipolar cells with the ganglion cells.

Ganglion Cells

The final layer of the retina contains the *ganglion cells*. The axons of the ganglion cells pass across the surface of the retina and collect in a bundle to form the optic nerve. It is interesting to note that each eye contains roughly 125 million receptors, but only 1 million ganglion cells. Clearly there is significant processing done in the retina to perform data reduction.

Visual Cortex

The axons of the ganglion cells, bundled into the optic nerve, connect to an area of the brain called the lateral geniculate nucleus, as illustrated in Figure 18.2. From this point the fibers fan out into the primary visual cortex, located at the back of the brain. The axons of the ganglion cells make synapses with lateral geniculate cells, and the axons of the lateral geniculate cells make synapses with cells in the visual cortex. The *visual cortex* is the region of the brain devoted to visual function and consists of many layers of cells.

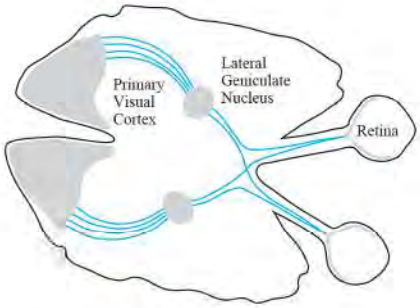


Figure 18.2 Visual Pathway

The connections along the visual pathway are far from random. The mapping from each layer to the next is highly organized. The axons from the ganglion cells in a certain part of the retina go to cells in a particular part of the lateral geniculate, which in turn go to a particular part of the visual cortex. (This topographic mapping was one of the inspirations for the self-organizing feature map described in Chapter 14.) In addition, as we can see in Figure 18.2, each hemisphere of the brain receives input from both eyes, since half of the optic nerve fibers cross and the other half stay uncrossed. It turns out that the left half of each visual field ends up in the right half of the brain, and the right half of each visual field ends up in the left half of the brain.

Illusions

We now have some idea of the general structure of the visual pathway, but how does it function? What is the purpose of the three layers of the retina? What operations are performed by the lateral geniculate? Some hints to the

answers to these questions can be obtained by investigating visual illusions.

Why are there so many visual illusions? Mechanisms that overcome imperfections of the retinal uptake process imply illusions. Grossberg and others have used the vast store of known illusions to probe adaptive perceptual mechanisms [GrMi89]. If we can develop mathematical models that produce the same illusions the biological system does, then we may have a mechanism that describes how this part of the brain works. To help us understand why illusions exist, we will first consider some of the imperfections of the retinal uptake process.

Optic Disk

Figure 18.3 is the view of the retina that an ophthalmologist has when looking into the eye through the cornea. The large pale circle is the *optic disk*, where the optic nerve leaves the retina on its way to the lateral geniculate. This is also where arteries enter and veins leave the retina. The optic disk causes a blind spot in our vision, as we will discuss in a moment.

Fovea

The darker disk to the right of the optic disk is the *fovea*, which constitutes the center of our field of vision. This is a region of the retina, about half a millimeter in diameter, that contains only cones. Although cones are distributed throughout the retina, they are most densely packed in the fovea. In addition, in this area of the retina the other layers are displaced to the side, so that the cones lie at the front. The densely packed photoreceptors, and the lack of obstruction, give us our best fine-detail vision at the fovea, which allows us to precisely focus the lens.

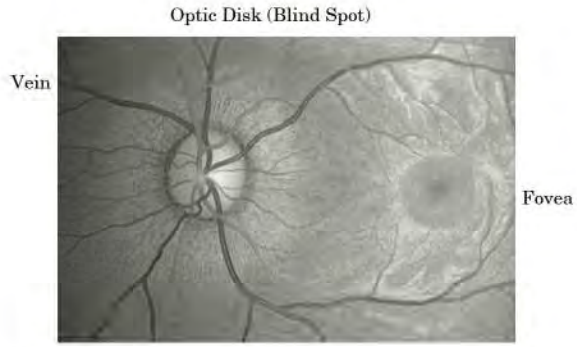


Figure 18.3 Back of the Eye (from [John01])

From Figure 18.3 we can see that there are a number of imperfections in retinal uptake. First, there are no rods and cones in the optic disk, which leaves a blind spot in our field of vision. We are not normally aware of the blind spot because of processing done in the visual pathway, but we can identify it with a simple test. Look at the blue circle on the left side of Figure 18.4, while covering your left eye. As you move your head closer to the page, then farther away, you will notice a point (about nine inches away)

at which the circle on the right will disappear from your field of vision. (You are still looking at the circle on the left.) If you haven't tried this before, it can be a little disconcerting. The interesting thing is that we don't see our blind spot as a black hole. Somehow our brains fill in the missing region.



Figure 18.4 Test for the Blind Spot

Other imperfections in the retinal uptake are the arteries and veins that cross in front of the photoreceptors at the back of the retina. These obstruct the rods and cones from receiving all of the light in the visual field. In addition, because the photoreceptors are at the back of the retina, light must pass through the other two layers to reach them.

Figure 18.5 illustrates the effect of these imperfections. Here we see an edge displayed on the retina. The drawing on the right illustrates the image initially perceived by the photoreceptors. The regions covered by the blind spot and the veins are not observed by the rods and cones. (The reason we do not "see" the arteries, veins, etc., is that the vision pathway does not respond to stabilized images. The eyeballs are constantly jerking, in what are called saccadic movements, so that even fixed objects in our field of vision are moving relative to the eye. The veins are fixed relative to the eyeball, so they fade from our vision.)

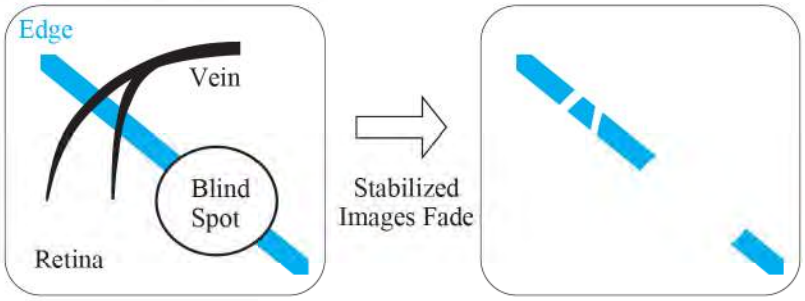


Figure 18.5 Perception of an Edge on the Retina (after [Gros90])

Emergent Segmentation
Featural Filling-in

Because we do not see edges as displayed on the right side of Figure 18.5, the neural systems in our visual pathway must be performing some operation that compensates for the distortions and completes the image. Grossberg suggests [GrMi89] that there are two primary types of compensatory processing involved. The first, which he calls *emergent segmentation*, completes missing boundaries. The second, which he calls *featural filling-in*, fills in the color and brightness inside the resulting boundaries. These two processes are illustrated in Figure 18.6. In the top figure we see an edge as it is originally perceived by the rods and cones, with missing sections. In

the lower figure we see the completed edge, after the emergent segmentation and featural filling-in.

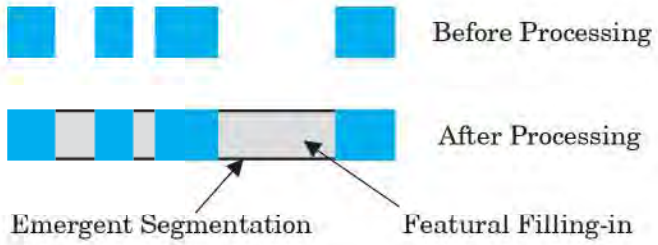
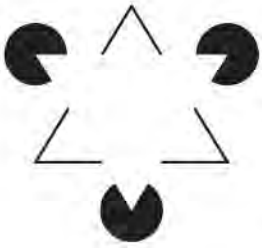
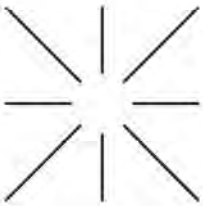


Figure 18.6 Compensatory Processing (after [Gros90])



If the processing along the visual pathway is recreating missing parts of the images we see, there must be times when it makes mistakes, since it cannot know exactly those parts of a scene from which it receives no light. These mistakes are illustrated by visual illusions. Consider, for example, the two figures in the left margin. In the top figure you should be able to see a bright white triangle lying on top of several other black objects. In fact, no such triangle exists in the figure. It is purely a creation of the emergent segmentation and featural filling-in process of your visual system. The same is true of the bright white circle which appears to lie on top of the lines in the lower-left figure.



The featural filling-in process is also demonstrated in Figure 18.7. This illusion is called neon color spreading [vanT75]. In the diagram on the right you may be able to see light blue diamonds, or even wide light blue lines criss-crossing the figure. In the diagram on the left you may be able to see a light blue ring. The blue you see filling in the diamonds and the ring is not a result of the color having been smeared during the printing process, nor is it caused by the scattering of light. This effect does not appear on the retina at all. It does not exist, except in your brain. (The perception of neon color spreading can vary from individual to individual, and the strength of the perception is dependent on the colors used. If you do not notice the effect in Figure 18.7, look at the cover of any issue of the journal *Neural Networks*, Pergamon Press.)

Later in this chapter we will discuss some neural network models that can help to explain the processes that implement emergent segmentation, as well as other visual phenomena.

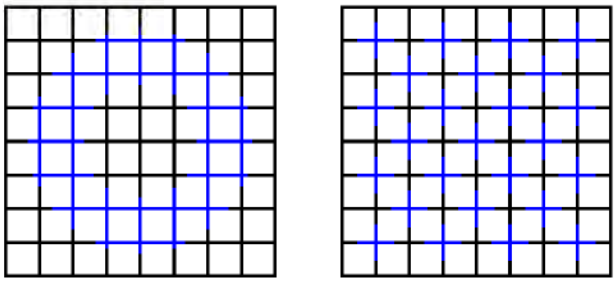


Figure 18.7 Neon Color Spreading (Featural Filling In)

Vision Normalization

Brightness Constancy
Brightness Contrast

In addition to emergent segmentation and featural filling-in, there are two other phenomena that give us an indication of what operations are being performed in the early vision system: *brightness constancy* and *brightness contrast*. The brightness constancy effect is evidenced by the test illustrated in Figure 18.8. In this test a subject is shown a small grey disk inside a darker grey annulus, which is illuminated by white light of a certain intensity. The subject is asked to indicate the brightness of the central disk by looking at a series of grey disks, separately illuminated, and selecting the disk with the same brightness. Next, the brightness of the light illuminating the grey disk and dark annulus is increased, and the subject is again asked to select the disk with the same brightness. This process is repeated for several different levels of illumination. It turns out that in each case the subject will choose the same disk as matching the original central disk. Even though the total light entering the subject's eye is 10 to 100 times brighter, it is only the relative brightness that registers.

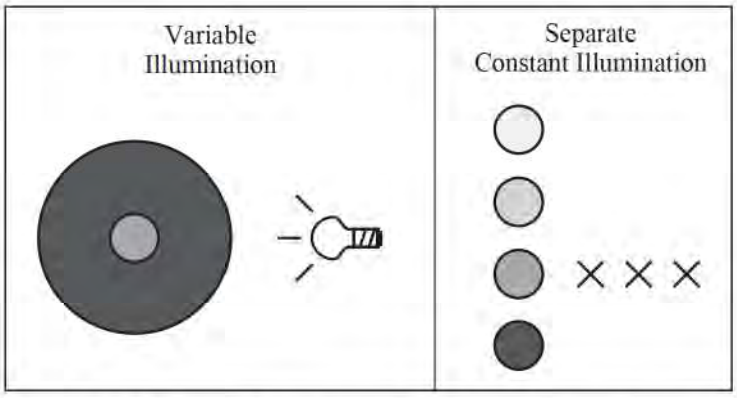


Figure 18.8 Test of Brightness Constancy (after [Gros90])



Another phenomenon of the vision system, which is closely related to brightness constancy, is brightness contrast. This effect is illustrated by the two figures in the left margin. At the centers of the two figures we have two small disks with equivalent grey scale. The small disk in the top figure is surrounded by a darker annulus, while the small disk in the lower figure is surrounded by a lighter annulus. Even though both disks have the same grey scale, the one inside the darker annulus appears brighter. This is because our vision system is sensitive to relative intensities. It would seem that the total activity across the image is held constant.

The properties of brightness constancy and brightness contrast are very important to our vision system. Since we see things in so many different lighting conditions, if we were not able to compensate for the absolute intensity of a scene, we would never learn to recognize things. Grossberg calls this process of normalization "discounting the illuminant."

In the rest of this chapter we want to present a neural network architecture that is consistent with the physical phenomena discussed in this section.

Basic Nonlinear Model

Before we introduce the Grossberg network, we will begin by looking at some of the building blocks that make up the network. The first building block is the *"leaky" integrator*, which is shown in Figure 18.9. The basic equation for this system is

Leaky Integrator

$$\varepsilon \frac{dn(t)}{dt} = -n(t) + p(t), \tag{18.1}$$

Time Constant

where ε is the system *time constant*.

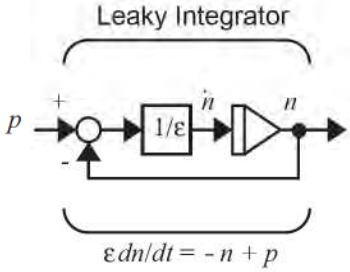


Figure 18.9 Leaky Integrator

The response of the leaky integrator to an arbitrary input $p(t)$ is

$$n(t) = e^{-t/\varepsilon} n(0) + \frac{1}{\varepsilon} \int_0^t e^{-(t-\tau)/\varepsilon} p(\tau) d\tau. \tag{18.2}$$

For example, if the input $p(t)$ is constant and the initial condition $n(0)$ is zero, Eq. (18.2) will produce

$$n(t) = p(1 - e^{-t/\varepsilon}). \tag{18.3}$$

A graph of this response, for $p = 1$ and $\varepsilon = 1$, is given in Figure 18.10. The response exponentially approaches a steady state value of 1.

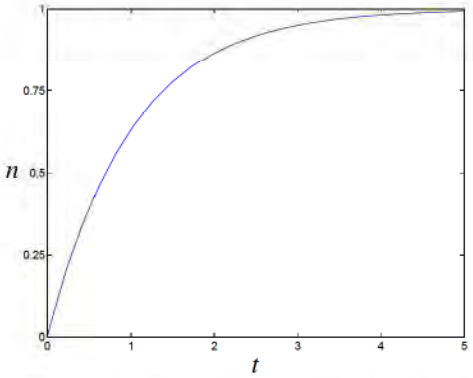


Figure 18.10 Leaky Integrator Response

There are two important properties of the leaky integrator that we want to note. First, because Eq. (18.1) is linear, if the input p is scaled, then the response $n(t)$ will be scaled by the same amount. For example, if the input is doubled, then the response will also be doubled, but will maintain the same shape. This is evident in Eq. (18.3). Second, the speed of response of the leaky integrator is determined by the time constant ε. When ε decreases, the response becomes faster; when ε increases, the response becomes slower. (See Problem P18.1.)



To experiment with the leaky integrator, use the Neural Network Design Demonstration Leaky Integrator (`nnd15li`).

Shunting Model

The leaky integrator forms the nucleus of one of Grossberg's fundamental neural models: the *shunting model*, which is shown in Figure 18.11. The equation of operation of this network is

$$\varepsilon \frac{dn(t)}{dt} = -n(t) + (b^+ - n(t))p^+ - (n(t) + b^-)p^-, \tag{18.4}$$

Excitatory

Inhibitory

where p^+ is a nonnegative value representing the *excitatory* input to the network (the input that causes the response to increase), and p^- is a nonnegative value representing the *inhibitory* input (the input that causes the response to decrease). The biases b^+ and b^- are nonnegative constants that determine the upper and lower limits on the neuron response, as we will explain next.

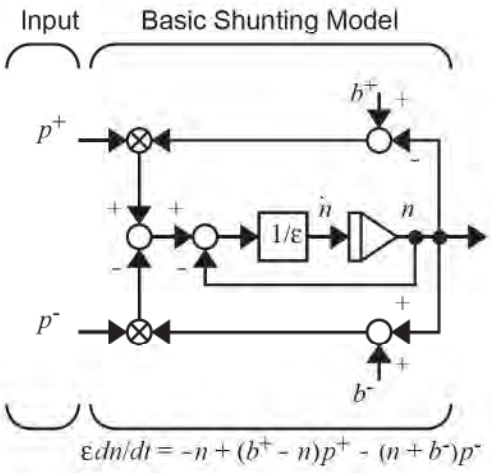
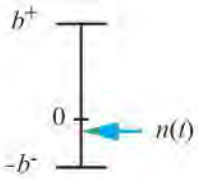


Figure 18.11 Shunting Model

There are three terms on the right-hand side of Eq. (18.4). When the net sign of these three terms is positive, $n(t)$ will go up. When the net sign is negative, $n(t)$ will go down. To understand the performance of the network, let's investigate the three terms individually.

The first term, $-n(t)$, is a linear decay term, which is also found in the leaky integrator. It is negative whenever $n(t)$ is positive, and positive whenever $n(t)$ is negative. The second term, $(b^+ - n(t))p^+$, provides nonlinear gain control. This term will be positive while $n(t)$ is less than b^+, but will become zero when $n(t) = b^+$. This effectively sets an upper limit on $n(t)$ of b^+. The third term, $-(n(t) + b^-)p^-$, also provides nonlinear gain control. It sets a lower limit on $n(t)$ of $-b^-$.

Figure 18.12 illustrates the performance of the shunting network when $b^+ = 1$, $b^- = 0$ and $\varepsilon = 1$. In the left graph we see the network response when the excitatory input $p^+ = 1$ and the inhibitory input $p^- = 0$. For the right graph $p^+ = 5$ and $p^- = 0$. Notice that even though the excitatory input is increased by a factor of 5, the steady state network response is increased by less than a factor of 2. If we were to continue to increase the excitatory input, we would find that the steady state network response would increase, but would always be less than $b^+ = 1$.



If we apply an inhibitory input to the shunting network, the steady state network response will decrease, but will remain greater than $-b^-$. To summarize the operation of the shunting model, if $n(0)$ falls between b^+ and $-b^-$, then $n(t)$ will remain between these limits, as shown in the figure in the left margin.

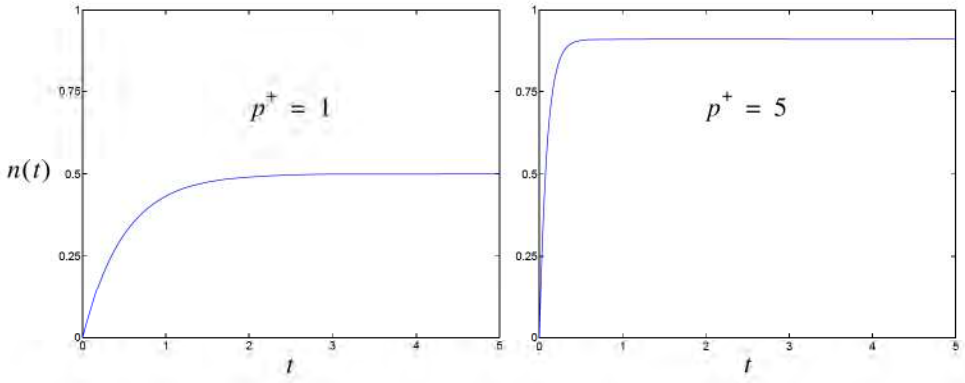


Figure 18.12 Shunting Network Response

The shunting model will form the basis for the Grossberg competitive network, which we will discuss in the next section. The nonlinear gain control will be used to normalize input patterns and to maintain relative intensities over a wide range of total intensity.



To experiment with the shunting network, use the Neural Network Design Demonstration Shunting Network (**nnd15sn**).

Two-Layer Competitive Network

We are now ready to present the Grossberg competitive network. This network was inspired by the operation of the mammalian visual system, which we discussed in the opening section of this chapter. (Grossberg was influenced by the work of Chistoph von der Malsburg [vond73], which was influenced in turn by the Nobel-prize-winning experimental work of David Hubel and Torsten Wiesel [HuWi62].) A block diagram of the network is shown in Figure 18.13.

Short-Term Memory
Long-Term Memory

There are three components to the Grossberg network: Layer 1, Layer 2 and the adaptive weights. Layer 1 is a rough model of the operation of the retina, while Layer 2 represents the visual cortex. These models do not fully explain the complexity of the human visual system, but they do illustrate a number of its characteristics. The network includes *short-term memory* (STM) and *long-term memory* (LTM) mechanisms, and performs adaptation, filtering, normalization and contrast enhancement. In the following subsections we will discuss the operation of each of the components of the network.

As we analyze the various elements of the Grossberg network, you will notice the similarity to the Kohonen competitive network of the previous chapter.

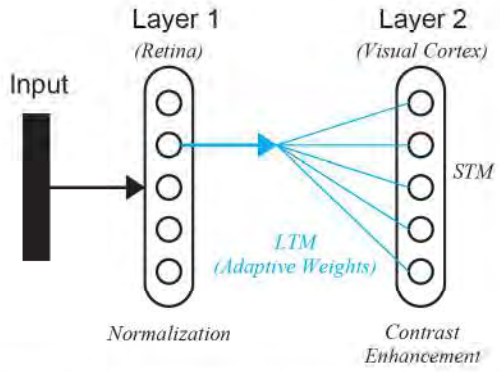


Figure 18.13 Grossberg Competitive Network

Layer 1

Layer 1 of the Grossberg network receives external inputs and normalizes the intensity of the input pattern. (Recall from Chapter 14 that the Kohonen network performs best when the input patterns are normalized. For the Grossberg network the normalization is accomplished by the first layer of the network.) A block diagram of this layer is given in Figure 18.14. Note that it uses the shunting model, with the excitatory and inhibitory inputs computed from the input vector $\mathbf{p}$.

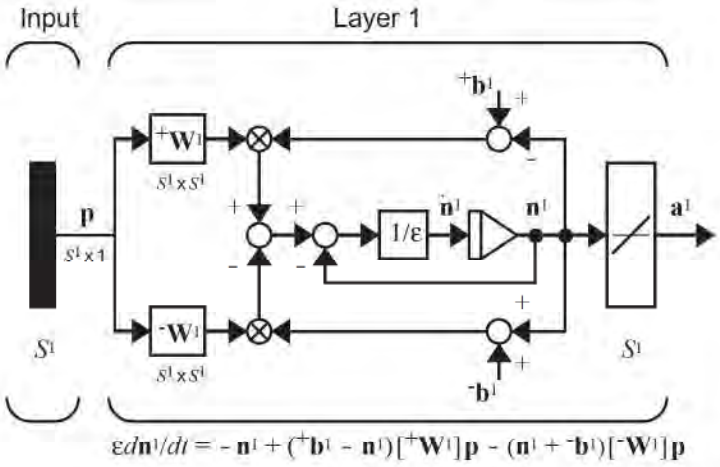


Figure 18.14 Layer 1 of the Grossberg Network

The equation of operation of Layer 1 is

$$\varepsilon \frac{d\mathbf{n}^1(t)}{dt} = -\mathbf{n}^1(t) + (^{+}\mathbf{b}^1 - \mathbf{n}^1(t))[^{+}\mathbf{W}^1]\mathbf{p} - (\mathbf{n}^1(t) + {}^{-}\mathbf{b}^1)[^{-}\mathbf{W}^1]\mathbf{p}. \qquad (18.5)$$

As we mentioned earlier, the parameter ε determines the speed of response. It is chosen so that the neuron responses will be much faster than the changes in the adaptive weights, which we will discuss in a later section.

Eq. (18.5) is a shunting model with excitatory input $[{}^{+}\mathbf{W}^{1}]\mathbf{p}$, where

$$ {}^{+}\mathbf{W}^{1} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{18.6} $$

Therefore the excitatory input to neuron i is the ith element of the input vector.

The inhibitory input to Layer 1 is $[{}^{-}\mathbf{W}^{1}]\mathbf{p}$, where

$$ {}^{-}\mathbf{W}^{1} = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix}. \tag{18.7} $$

Thus the inhibitory input to neuron i is the sum of all elements of the input vector, except the ith element.

On-Center/Off-Surround

The connection pattern defined by the matrices ${}^{+}\mathbf{W}^{1}$ and ${}^{-}\mathbf{W}^{1}$ is called an *on-center/off-surround* pattern. This is because the excitatory input for neuron i (which turns the neuron on) comes from the element of the input vector centered at the same location (element i), while the inhibitory input (which turns the neuron off) comes from surrounding locations. This type of connection pattern produces a normalization of the input pattern, as we will show in the following discussion.

For simplicity, we will set the inhibitory bias ${}^{-}\mathbf{b}^{1}$ to zero, which sets the lower limit of the shunting model to zero, and we will set all elements of the excitatory bias ${}^{+}\mathbf{b}^{1}$ to the same value, i.e.,

$$ {}^{+}b_{i}^{1} = {}^{+}b^{1}, \; i = 1, 2, \ldots, S^{1}, \tag{18.8} $$

so that the upper limit for all neurons will be the same.

To investigate the normalization effect of Layer 1, consider the response of neuron i:

$$\varepsilon \frac{dn_i^1(t)}{dt} = -n_i^1(t) + ({}^+b^1 - n_i^1(t))p_i - n_i^1(t)\sum_{j \neq i} p_j . \tag{18.9}$$

In the steady state ($dn_i^1(t)/dt = 0$) we have

$$0 = -n_i^1 + ({}^+b^1 - n_i^1)p_i - n_i^1\sum_{j \neq i} p_j . \tag{18.10}$$

If we solve for the steady state neuron output n_i^1 we find

$$n_i^1 = \frac{{}^+b^1 p_i}{1 + \sum_{j=1}^{S^1} p_j} . \tag{18.11}$$

We now define the relative intensity of input i to be

$$\bar{p}_i = \frac{p_i}{P} \text{ where } P = \sum_{j=1}^{S^1} p_j . \tag{18.12}$$

Then the steady state neuron activity can be written

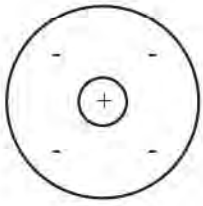
$$n_i^1 = \left(\frac{{}^+b^1 P}{1+P}\right)\bar{p}_i . \tag{18.13}$$

Therefore n_i^1 will be proportional to the relative intensity $\bar{p}_i$, regardless of the magnitude of the total input P. In addition, the total neuron activity is bounded:

$$\sum_{j=1}^{S^1} n_j^1 = \sum_{j=1}^{S^1} \left(\frac{{}^+b^1 P}{1+P}\right)\bar{p}_j = \left(\frac{{}^+b^1 P}{1+P}\right) \leq {}^+b^1 . \tag{18.14}$$

The input vector is normalized so that the total activity is less than ${}^+b^1$, while the relative intensities of the individual elements of the input vector are maintained. Therefore, the outputs of Layer 1, n_i^1, code the relative input intensities, $\bar{p}_i$, rather than the instantaneous fluctuations in the total input activity, P. This result is produced by the on-center/off-surround connection pattern of the inputs and the nonlinear gain control of the shunting model.

Note that Layer 1 of the Grossberg network explains the brightness constancy and brightness contrast characteristics of the human visual system, which we discussed on page 15-8. The network is sensitive to the relative



intensities of an image, rather than absolute intensities. In addition, experimental evidence has shown that the on-center/off-surround connection pattern is a characteristic feature of the receptive fields of retinal ganglion cells [Hube88]. (The receptive field is an area of the retina in which the photoreceptors feed into a given cell. The figure in the left margin illustrates the on-center/off-surround receptive field of a typical retinal ganglion cell. A "+" indicates an excitatory region, and a "-" indicates an inhibitory region. It is a two-dimensional pattern, as opposed to the one-dimensional connections of Eq. (18.6) and Eq. (18.7).)



To illustrate the performance of Layer 1, consider the case of two neurons, with ${}^{+}b^{1} = 1$, $\varepsilon = 0.1$:

$$(0.1)\frac{dn_1^1(t)}{dt} = -n_1^1(t) + (1 - n_1^1(t))p_1 - n_1^1(t)p_2, \tag{18.15}$$

$$(0.1)\frac{dn_2^1(t)}{dt} = -n_2^1(t) + (1 - n_2^1(t))p_2 - n_2^1(t)p_1. \tag{18.16}$$

The response of this network, for two different input vectors, is shown in Figure 18.15. For both input vectors, the second element is four times as large as the first element, although the total intensity of the second input vector is five times as large as that of the first input vector (50 vs. 10). From Figure 18.15 we can see that the response of the network maintains the relative intensities of the inputs, while limiting the total response. The total response $(n_1^1(t) + n_2^1(t))$ will always be less than 1.

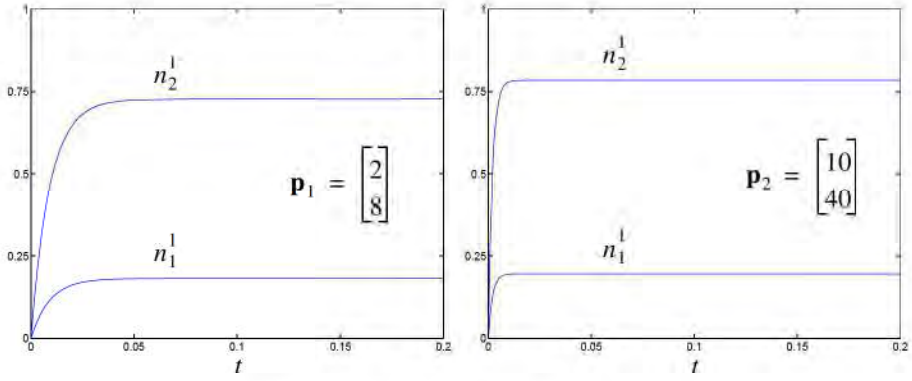


Figure 18.15 Layer 1 Response



To experiment with Layer 1 of the Grossberg network, use the Neural Network Design Demonstration Grossberg Layer 1 (`nnd15gl1`).

Layer 2

Short-Term Memory

Layer 2 of the Grossberg network, which is a layer of continuous-time instars, performs several functions. First, like Layer 1, it normalizes total activity in the layer. Second, it contrast enhances its pattern, so that the neuron that receives the largest input will dominate the response. (This is closely related to the winner-take-all competition in the Hamming network and the Kohonen network.) Finally, it operates as a *short-term memory* (STM) by storing the contrast-enhanced pattern.

Figure 18.16 is a diagram of Layer 2. As with Layer 1, the shunting model forms the basis for Layer 2. The main difference between Layer 2 and Layer 1 is that Layer 2 uses feedback connections. The feedback enables the network to store a pattern, even after the input has been removed. The feedback also performs the competition that causes the contrast enhancement of the pattern. We will demonstrate these properties in the following discussion.

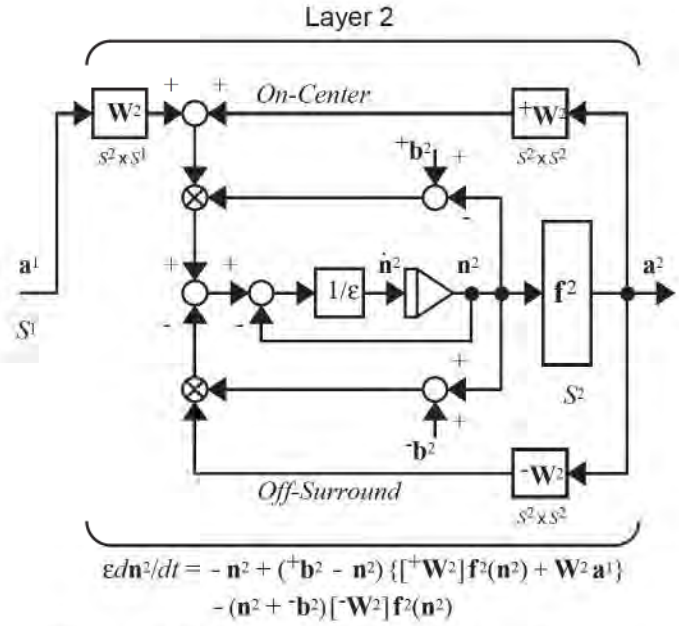


Figure 18.16 Layer 2 of the Grossberg Network

The equation of operation of Layer 2 is

$$\varepsilon \frac{d\mathbf{n}^2(t)}{dt} = -\mathbf{n}^2(t) + ({}^{+}\mathbf{b}^2 - \mathbf{n}^2(t))\{[{}^{+}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^2\mathbf{a}^1\} - (\mathbf{n}^2(t) + {}^{-}\mathbf{b}^2)[{}^{-}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) \quad (18.17)$$

This is a shunting model with excitatory input $\{[{}^{+}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^2\mathbf{a}^1\}$, where ${}^{+}\mathbf{W}^2 = {}^{+}\mathbf{W}^1$ provides on-center feedback connections, and $\mathbf{W}^2$ consists of adaptive weights, analogous to the weights in the Kohonen network. The rows of $\mathbf{W}^2$, after training, will represent the prototype patterns. The inhibitory input to the shunting model is $[{}^{-}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t))$, where ${}^{-}\mathbf{W}^2 = {}^{-}\mathbf{W}^1$ provides off-surround feedback connections.



To illustrate the performance of Layer 2, consider a two-neuron layer with

$$\varepsilon = 0.1,\ {}^{+}\mathbf{b}^2 = \begin{bmatrix}1\\1\end{bmatrix},\ {}^{-}\mathbf{b}^2 = \begin{bmatrix}0\\0\end{bmatrix},\ \mathbf{W}^2 = \begin{bmatrix}({}_1\mathbf{w}^2)^T\\({}_2\mathbf{w}^2)^T\end{bmatrix} = \begin{bmatrix}0.9 & 0.45\\0.45 & 0.9\end{bmatrix}, \quad (18.18)$$

and

$$f^2(n) = \frac{10(n)^2}{1+(n)^2}. \quad (18.19)$$

The equations of operation of the layer will be

$$(0.1)\frac{dn_1^2(t)}{dt} = -n_1^2(t) + (1-n_1^2(t))\{f^2(n_1^2(t)) + ({}_1\mathbf{w}^2)^T\mathbf{a}^1\} - n_1^2(t)f^2(n_2^2(t)) \quad (18.20)$$

$$(0.1)\frac{dn_2^2(t)}{dt} = -n_2^2(t) + (1-n_2^2(t))\{f^2(n_2^2(t)) + ({}_2\mathbf{w}^2)^T\mathbf{a}^1\} - n_2^2(t)f^2(n_1^2(t)). \quad (18.21)$$

Note the relationship between these equations and the Hamming and Kohonen networks. The inputs to Layer 2 are the inner products between the prototype patterns (rows of the weight matrix $\mathbf{W}^2$) and the output of Layer 1 (normalized input pattern). The largest inner product will correspond to the prototype pattern closest to the input pattern. Layer 2 then performs a competition between the neurons, which tends to *contrast enhance* the output pattern — maintaining large outputs while attenuating small outputs. This contrast enhancement is generally milder than the winner-take-all competition of the Hamming and Kohonen networks. In the Hamming and Kohonen networks, the competition drives all but one of the neuron outputs to zero. The exception is the one with the largest input. In the Grossberg network, the competition maintains large values and attenuates small values, but does not necessarily drive all small values to zero. The amount

Contrast Enhance

of contrast enhancement is determined by the transfer function f^2, as we will see in the next section.

Figure 18.17 illustrates the response of Layer 2 when the input vector $\mathbf{a}^1 = [0.2\ 0.8]^T$ (the steady state result obtained from our Layer 1 example) is applied for 0.25 seconds and then removed.

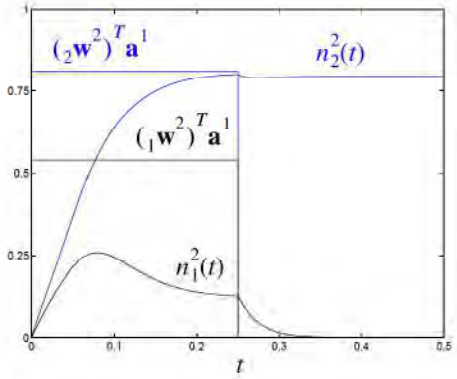


Figure 18.17 Layer 2 Response

There are two important characteristics of this response. First, even before the input is removed, some contrast enhancement is performed. The inputs to Layer 2 are

$$({}_1\mathbf{w}^2)^T\mathbf{a}^1 = [0.9\ 0.45]\begin{bmatrix}0.2\\0.8\end{bmatrix} = 0.54, \tag{18.22}$$

$$({}_2\mathbf{w}^2)^T\mathbf{a}^1 = [0.45\ 0.9]\begin{bmatrix}0.2\\0.8\end{bmatrix} = 0.81. \tag{18.23}$$

Therefore the second neuron has 1.5 times as much input as the first neuron. However, after 0.25 seconds the output of the second neuron is 6.34 times the output of the first neuron. The contrast between high and low has been increased dramatically.

The second characteristic of the response is that after the input has been set to zero, the network further enhances the contrast and stores the pattern. In Figure 18.17 we can see that after the input is removed (at 0.25 seconds) the output of the first neuron decays to zero, while the output of the second neuron reaches a steady state value of 0.79. This output is maintained, even after the input is removed. (Grossberg calls this behavior *reverberation* [Gross76].) It is the nonlinear feedback that enables the net-

work to store the pattern, and the on-center/off-surround connection pattern (determined by ${}^{+}\mathbf{W}^{2}$ and ${}^{-}\mathbf{W}^{2}$) that causes the contrast enhancement.

As an aside, note that we have used the on-center/off-surround structure in both layers of the Grossberg network. There are other connection patterns that could be used for different applications. Recall, for instance, the emergent segmentation problem discussed earlier in this chapter. A structure that has been proposed to implement this mechanism is the *oriented receptive field* [GrMi89], which is shown in the left margin. For this structure, the "on" (excitatory) connections come from one side of the field (indicated by the blue region), and the "off" (inhibitory) connections come from the other side of the field (indicated by the white region).

Oriented Receptive Field



The operation of the oriented receptive field is illustrated in Figure 18.18. When the field is aligned with an edge, the corresponding neuron is activated (large response). When the field is not aligned with an edge, then the neuron is inactive (small response). This explains why we might perceive an edge where none exists, as can be seen in the right-most receptive field shown in Figure 18.18.

For a complete discussion of oriented receptive fields and how they can be incorporated into a neural network architecture for preattentive vision, see [GrMi89]. This paper also discusses a mechanism for featural filling-in.

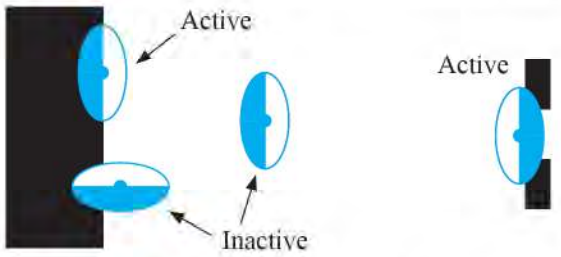
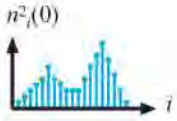


Figure 18.18 Operation of Oriented Receptive Field

Choice of Transfer Function



The behavior of Layer 2 of the Grossberg network depends very much on the transfer function $f^2(n)$. For example, suppose that an input has been applied for some length of time, so that the output has stabilized to the pattern shown in the left margin. (Each point represents the output of an individual neuron.) If the input is then removed, Figure 18.19 demonstrates how the choice of $f^2(n)$ will affect the steady state response of the network. (See [Gross82].)

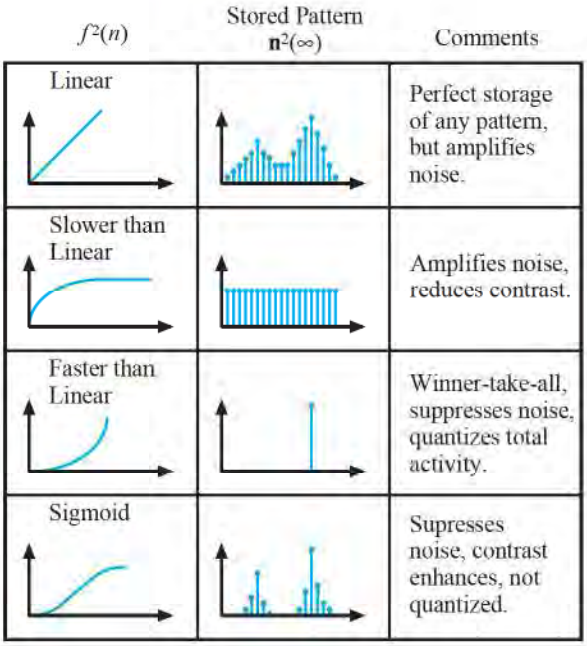


Figure 18.19 Effect of Transfer Function $f^2(n)$ (after [Gross82])

If the transfer function is *linear*, the pattern is perfectly stored. Unfortunately, the noise in the pattern will be amplified and stored as easily as the significant inputs. (See Problem P18.6.) If the transfer function is *slower-than-linear* (e.g., $f^2(n) = 1 - e^{-n}$), the steady state response is independent of the initial conditions; all neurons that begin with nonzero values will come to the same level in the steady state. All contrast is eliminated, and noise is amplified.

Faster-than-linear transfer functions (e.g., $f^2(n) = (n)^2$) produce a winner-take-all competition. Only those neurons with the largest initial values are stored; all others are driven to zero. This minimizes the effect of noise, but quantizes the response into an all-or-nothing signal (as in the Hamming and Kohonen networks).

A *sigmoid* function is faster-than-linear for small signals, approximately linear for intermediate signals and slower-than-linear for large signals. When a sigmoid transfer function is used in Layer 2, the pattern is contrast enhanced; larger values are amplified, and smaller values are attenuated. All initial neuron outputs that are less than a certain level (called the *quenching threshold* by Grossberg [Gros76]) decay to zero. This merges the noise suppression of the faster-than-linear transfer functions with the perfect storage produced by linear transfer functions.



To experiment with Layer 2 of the Grossberg network, use the Neural Network Design Demonstration Grossberg Layer 2 (`nnd15gl2`).

Learning Law

Long-Term Memory

The third component of the Grossberg network is the learning law for the adaptive weights $\mathbf{W}^2$. Grossberg calls these adaptive weights the *long-term memory* (LTM). This is because the rows of $\mathbf{W}^2$ will represent patterns that have been stored and that the network will be able to recognize. As in the Kohonen and Hamming networks, the stored pattern that is closest to an input pattern will produce the largest output in Layer 2. In the next subsection we will look more closely at the relationship between the Grossberg network and the Kohonen network.

One learning law for $\mathbf{W}^2$ is given by

$$\frac{dw_{i,j}^2(t)}{dt} = \alpha\{-w_{i,j}^2(t) + n_i^2(t)n_j^1(t)\}. \tag{18.24}$$

The first term in the bracket on the right-hand side of Eq. (18.24) is a passive decay term, which we have seen in the Layer 1 and Layer 2 equations, while the second term implements a Hebbian-type learning. Together, these terms implement the Hebb rule with decay, which was discussed in Chapter 13.

Recall from Chapter 13 that it is often useful to turn off learning (and forgetting) when $n_i^2(t)$ is not active. This can be accomplished by the following learning law:

$$\frac{dw_{i,j}^2(t)}{dt} = \alpha n_i^2(t)\{-w_{i,j}^2(t) + n_j^1(t)\}, \tag{18.25}$$

or, in vector form,

$$\frac{d[{}_i\mathbf{w}^2(t)]}{dt} = \alpha n_i^2(t)\{-[{}_i\mathbf{w}^2(t)] + \mathbf{n}^1(t)\}, \tag{18.26}$$

where ${}_i\mathbf{w}^2(t)$ is a vector composed of the elements of the ith row of $\mathbf{W}^2$ (see Eq. (4.4)).

The terms on the right-hand side of Eq. (18.25) are multiplied (gated) by $n_i^2(t)$, which allows learning (and forgetting) to occur only when $n_i^2(t)$ is not zero. This is the continuous-time implementation of the instar learning rule, which we introduced in Chapter 13 (Eq. (15.32)). In the following subsection we will demonstrate the equivalence of Eq. (18.25) and Eq. (15.32).



To illustrate the performance of the Grossberg learning law, consider a network with two neurons in each layer. The weight update equations would be

$$\frac{dw^2_{1,1}(t)}{dt} = n^2_1(t)\{-w^2_{1,1}(t) + n^1_1(t)\}, \tag{18.27}$$

$$\frac{dw^2_{1,2}(t)}{dt} = n^2_1(t)\{-w^2_{1,2}(t) + n^1_2(t)\}, \tag{18.28}$$

$$\frac{dw^2_{2,1}(t)}{dt} = n^2_2(t)\{-w^2_{2,1}(t) + n^1_1(t)\}, \tag{18.29}$$

$$\frac{dw^2_{2,2}(t)}{dt} = n^2_2(t)\{-w^2_{2,2}(t) + n^1_2(t)\}, \tag{18.30}$$

where the learning rate coefficient α has been set to 1. To simplify our example, we will assume that two different input patterns are alternately presented to the network for periods of 0.2 seconds at a time. We will also assume that Layer 1 and Layer 2 converge very quickly, in comparison with the convergence of the weights, so that the neuron outputs are effectively constant over the 0.2 seconds. The Layer 1 and Layer 2 outputs for the two different input patterns will be

$$\text{for pattern 1: } \mathbf{n}^1 = \begin{bmatrix} 0.9 \\ 0.45 \end{bmatrix}, \mathbf{n}^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{18.31}$$

$$\text{for pattern 2: } \mathbf{n}^1 = \begin{bmatrix} 0.45 \\ 0.9 \end{bmatrix}, \mathbf{n}^2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{18.32}$$

Pattern 1 is coded by the first neuron in Layer 2, and pattern 2 is coded by the second neuron in Layer 2.

Figure 18.20 illustrates the response of the adaptive weights, beginning with all weights set to zero. Note that the first row of the weight matrix ($w^2_{1,1}(t)$ and $w^2_{1,2}(t)$) is only adjusted during those periods when $n^2_1(t)$ is nonzero, and that it converges to the corresponding $\mathbf{n}^1$ pattern ($n^1_1(t) = 0.9$ and $n^1_2(t) = 0.45$). (The elements in the first row of the weight matrix are indicated by the blue lines in Figure 18.20.) Also, the second row of the weight matrix ($w^2_{2,1}(t)$ and $w^2_{2,2}(t)$) is only adjusted during those periods when $n^2_2(t)$ is nonzero, and it converges to the corresponding $\mathbf{n}^1$ pattern ($n^1_1(t) = 0.45$ and $n^1_2(t) = 0.9$). (The elements in the second row of the weight matrix are indicated by the black lines in Figure 18.20.)

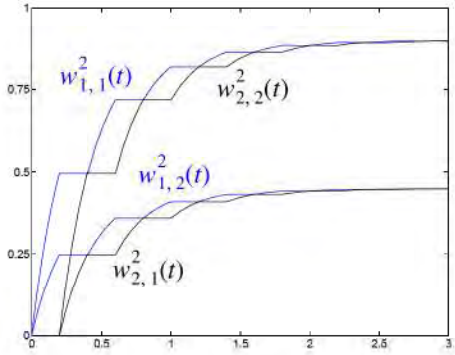


Figure 18.20 Response of the Adaptive Weights



To experiment with the adaptive weights, use the Neural Network Design Demonstration Adaptive Weights (**nnd15aw**).

Relation to Kohonen Law

In the previous section we indicated that the Grossberg learning law was a continuous-time version of the instar learning law, which we discussed in Chapter 13. Now we want to demonstrate this fact. We will also show that the Grossberg network is, in its simplest form, a continuous-time version of the Kohonen competitive network of Chapter 14.

To begin, let's repeat the Grossberg learning law of Eq. (18.25):

$$\frac{d[{}_i\mathbf{w}^2(t)]}{dt} = \alpha n_i^2(t)\{-[{}_i\mathbf{w}^2(t)] + \mathbf{n}^1(t)\}. \tag{18.33}$$

If we approximate the derivative by

$$\frac{d[{}_i\mathbf{w}^2(t)]}{dt} \approx \frac{{}_i\mathbf{w}^2(t+\Delta t) - {}_i\mathbf{w}^2(t)}{\Delta t}, \tag{18.34}$$

then we can rewrite Eq. (18.33) as

$${}_i\mathbf{w}^2(t+\Delta t) = {}_i\mathbf{w}^2(t) + \alpha(\Delta t)n_i^2(t)\{-{}_i\mathbf{w}^2(t) + \mathbf{n}^1(t)\}. \tag{18.35}$$

(Compare this equation with the instar rule that was presented in Chapter 13 in Eq. (15.33).) If we rearrange terms, this can be reduced to

$${}_i\mathbf{w}^2(t+\Delta t) = \{1 - \alpha(\Delta t)n_i^2(t)\}{}_i\mathbf{w}^2(t) + \alpha(\Delta t)n_i^2(t)\{\mathbf{n}^1(t)\}. \tag{18.36}$$

To simplify the analysis further, assume that a faster-than-linear transfer function is used in Layer 2, so that only one neuron in that layer will have

a nonzero output; call it neuron i^*. Then only row i^* of the weight matrix will be updated:

$$_{i^*}\mathbf{w}^2(t+\Delta t) = \{1-\alpha'\}_{i^*}\mathbf{w}^2(t) + \{\alpha'\}\mathbf{n}^1(t), \tag{18.37}$$

where $\alpha' = \alpha(\Delta t)n^2_{i^*}(t)$.

This is almost identical to the Kohonen rule for the competitive network that we introduced in Chapter 14 in Eq. (16.13). The weight vector for the winning neuron (with nonzero output) will be moved toward $\mathbf{n}^1$, which is a normalized version of the current input pattern.

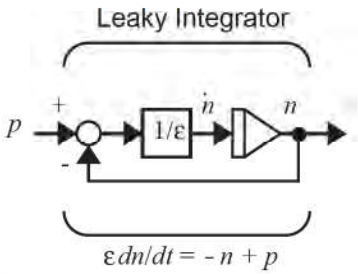
There are three major differences between the Grossberg network that we have presented in this chapter and the basic Kohonen competitive network. First, the Grossberg network is a continuous-time network (satisfies a set of nonlinear differential equations). Second, Layer 1 of the Grossberg network automatically normalizes the input vectors. Third, Layer 2 of the Grossberg network can perform a "soft" competition, rather than the winner-take-all competition of the Kohonen network. This soft competition allows more than one neuron in Layer 2 to learn. This causes the Grossberg network to operate as a feature map.

Summary of Results

Basic Nonlinear Model

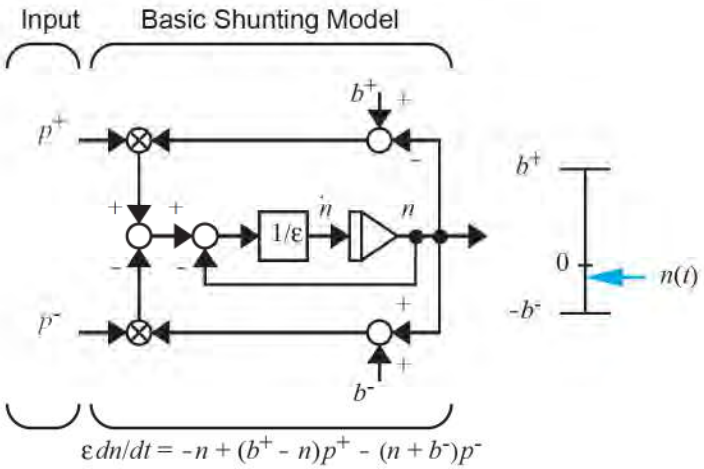
Leaky Integrator

$$\varepsilon \frac{dn(t)}{dt} = -n(t) + p(t)$$

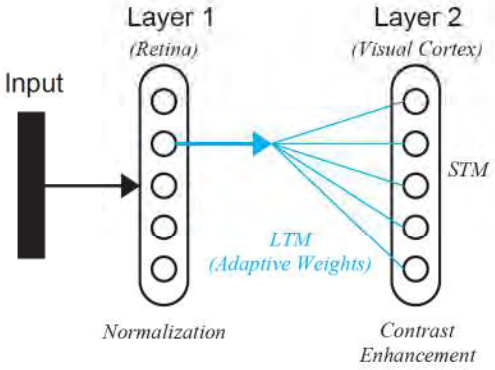


Shunting Model

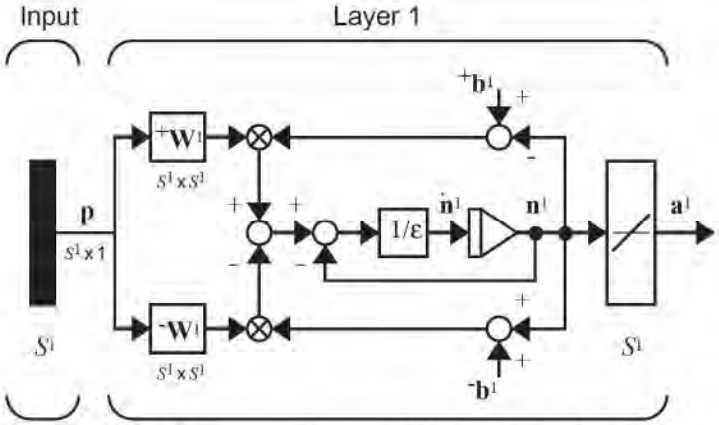
$$\varepsilon \frac{dn(t)}{dt} = -n(t) + (b^+ - n(t))p^+ - (n(t) + b^-)p^-$$



Two-Layer Competitive Network



Layer 1



$$\varepsilon d\mathbf{n}^1/dt = -\mathbf{n}^1 + ({}^+\mathbf{b}^1 - \mathbf{n}^1)[{}^+\mathbf{W}^1]\mathbf{p} - (\mathbf{n}^1 + {}^-\mathbf{b}^1)[{}^-\mathbf{W}^1]\mathbf{p}$$

$$\varepsilon\frac{d\mathbf{n}^1(t)}{dt} = -\mathbf{n}^1(t) + ({}^+\mathbf{b}^1 - \mathbf{n}^1(t))[{}^+\mathbf{W}^1]\mathbf{p} - (\mathbf{n}^1(t) + {}^-\mathbf{b}^1)[{}^-\mathbf{W}^1]\mathbf{p}$$

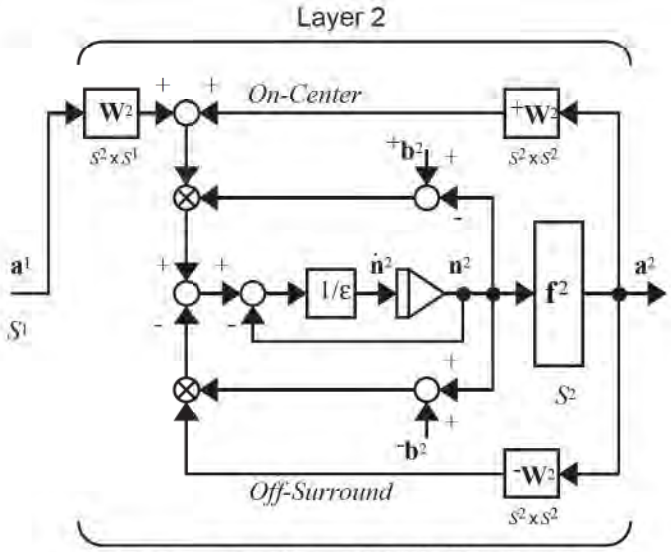
$${}^+\mathbf{W}^1 = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \qquad {}^-\mathbf{W}^1 = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix}$$

On-Center *Off-Surround*

Steady State Neuron Activity

$$n_i^1 = \left(\frac{{}^+b^1 P}{1+P}\right)\bar{p}_i, \text{ where } \bar{p}_i = \frac{p_i}{P} \text{ and } P = \sum_{j=1}^{S^1} p_j$$

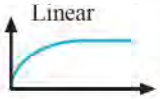
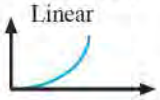
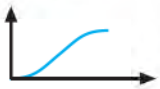
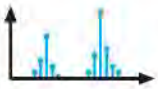
Layer 2



$$\varepsilon d\mathbf{n}^2/dt = -\mathbf{n}^2 + ({}^+\mathbf{b}^2 - \mathbf{n}^2)\{[{}^+\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2) + \mathbf{W}^2\mathbf{a}^1\} - (\mathbf{n}^2 + {}^-\mathbf{b}^2)[{}^-\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2)$$

$$\varepsilon\frac{d\mathbf{n}^2(t)}{dt} = -\mathbf{n}^2(t) + ({}^+\mathbf{b}^2 - \mathbf{n}^2(t))\{[{}^+\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^2\mathbf{a}^1\} - (\mathbf{n}^2(t) + {}^-\mathbf{b}^2)[{}^-\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t))$$

Choice of Transfer Function

$f^2(n)$	Stored Pattern $\mathbf{n}^2(\infty)$	Comments
Linear 		Perfect storage of any pattern, but amplifies noise.
Slower than Linear 		Amplifies noise, reduces contrast.
Faster than Linear 		Winner-take-all, suppresses noise, quantizes total activity.
Sigmoid 		Supresses noise, contrast enhances, not quantized.

Learning Law

$$\frac{d[{}_i\mathbf{w}^2(t)]}{dt} = \alpha n_i^2(t)\{-[{}_i\mathbf{w}^2(t)] + \mathbf{n}^1(t)\}$$

(Continuous-Time Instar Learning)

Solved Problems

P18.1 Demonstrate the effect of the coefficient ε on the performance of the leaky integrator, which is shown in Figure P18.1, with the input $p = 1$.

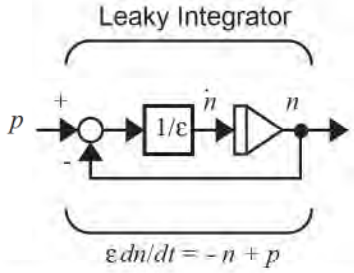


Figure P18.1 Leaky Integrator

The equation of operation for the leaky integrator is

$$\varepsilon \frac{dn(t)}{dt} = -n(t) + p(t).$$

The solution to this differential equation, for an arbitrary input $p(t)$, is

$$n(t) = e^{-t/\varepsilon} n(0) + \frac{1}{\varepsilon} \int_0^t e^{-(t-\tau)/\varepsilon} p(t-\tau) d\tau .$$

If $p(t) = 1$, the solution will be

$$n(t) = e^{-t/\varepsilon} n(0) + \frac{1}{\varepsilon} \int_0^t e^{-(t-\tau)/\varepsilon} d\tau .$$

We want to show how this response changes as a function of ε. The response will be

$$n(t) = e^{-t/\varepsilon} n(0) + (1 - e^{-t/\varepsilon}) = e^{-t/\varepsilon}(n(0) - 1) + 1 .$$

This response begins at $n(0)$, and then grows exponentially (or decays exponentially, depending on whether or not $n(0)$ is greater than or less than 1), approaching the steady state response of $n(\infty) = 1$. As ε is decreased, the response becomes faster (since $e^{-t/\varepsilon}$ decays more quickly), while the steady state value remains constant. Figure P18.2 illustrates the responses for $\varepsilon = 1, 0.5, 0.25, 0.125$, with $n(0) = 0$. Notice that the steady state value remains 1 for each case. Only the speed of response changes.

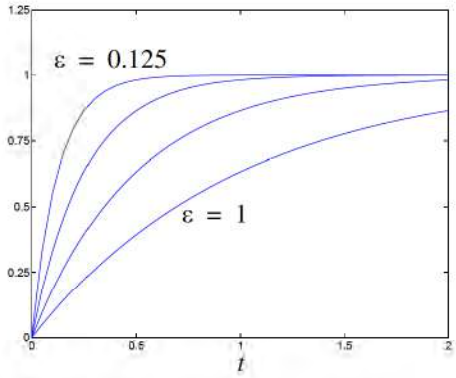


Figure P18.2 Effect of ε on Leaky Integrator Response

P18.2 Again using the leaky integrator of Figure P18.1, set $\varepsilon = 1$.

i. Find a difference equation approximation to the leaky integrator differential equation by approximating the derivative using

$$\frac{dn(t)}{dt} \approx \frac{n(t+\Delta t) - n(t)}{\Delta t}.$$

ii. Using $\Delta t = 0.1$, compare the response of this difference equation with the response of the differential equation for $p(t) = 1$ and $n(0) = 0$. Compare the two over the range $0 < t < 1$.

iii. Using the difference equation model for the leaky integrator, show that the response is a weighted average of previous inputs.

i. If we make the approximation to the derivative, we find

$$\frac{n(t+\Delta t) - n(t)}{\Delta t} = -n(t) + p(t)$$

or

$$n(t+\Delta t) = n(t) + \Delta t\{-n(t) + p(t)\} = (1-\Delta t)n(t) + (\Delta t)p(t).$$

ii. If we let $\Delta t = 0.1$ we obtain the difference equation

$$n(t+0.1) = 0.9n(t) + 0.1p(t).$$

If we let $p(t) = 1$ and $n(0) = 0$, then we can solve for $n(t)$:

$$n(0.1) = 0.9n(0) + 0.1p(0) = 0.1$$

$$n(0.2) = 0.9n(0.1) + 0.1p(0.1) = 0.9(0.1) + 0.1(1) = 0.19,$$

$$n(0.3) = 0.9n(0.2) + 0.1p(0.2) = 0.9(0.19) + 0.1(1) = 0.271,$$

$$n(0.4) = 0.9n(0.3) + 0.1p(0.3) = 0.9(0.271) + 0.1(1) = 0.3439,$$

$$n(0.5) = 0.9n(0.4) + 0.1p(0.4) = 0.9(0.3439) + 0.1(1) = 0.4095,$$

$$n(0.6) = 0.4686,\ n(0.7) = 0.5217,\ n(0.8) = 0.5695,$$

$$n(0.9) = 0.6126,\ n(1.0) = 0.6513.$$

From Problem P18.1, the solution to the differential equation is

$$n(t) = e^{-t/\varepsilon}n(0) + (1 - e^{-t/\varepsilon}) = (1 - e^{-t}).$$

Figure P18.3 illustrates the relationship between the difference equation solution and the differential equation solution. The black line represents the differential equation solution, and the blue circles represent the difference equation solution. The two solutions are very close, and can be made arbitrarily close by decreasing the interval Δt.

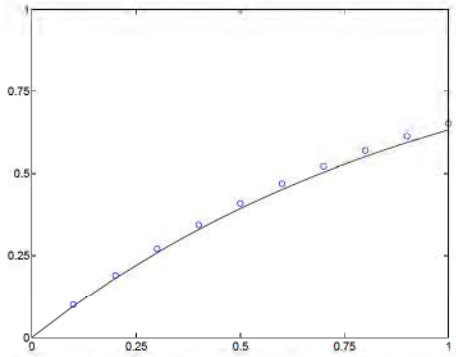


Figure P18.3 Comparison of Difference and Differential Equations

iii. Consider again the difference equation model of the leaky integrator, which we developed in part (ii):

$$n(t + 0.1) = 0.9n(t) + 0.1p(t).$$

If we start from a zero initial condition we find

$$n(0.1) = 0.9n(0) + 0.1p(0) = 0.1p(0),$$

$$n(0.2) = 0.9n(0.1) + 0.1p(0.1) = 0.9\{0.1p(0)\} + 0.1p(0.1) = 0.09p(0) + 0.1p(0.1)$$

$$n(0.3) = 0.9n(0.2) + 0.1p(0.2) = 0.081p(0) + 0.09p(0.1) + 0.1p(0.2)$$

$$\vdots$$

$$n(k0.1) = 0.1\{(0.9)^{k-1}p(0) + (0.9)^{k-2}p(0.1) + \cdots + p((k-1)0.1)\}.$$

Therefore the response of the leaky integrator is a weighted average of previous inputs, $p(0), p(0.1), \ldots, p((k-1)0.1)$. Note that the recent inputs contribute more to the response than the early inputs.

P18.3 Find the response of the shunting network shown in Figure P18.4 for $\varepsilon = 1$, $b^+ = 1$, $b^- = 1$, $p^+ = 0$, $p^- = 10$ and $n(0) = 0.5$.

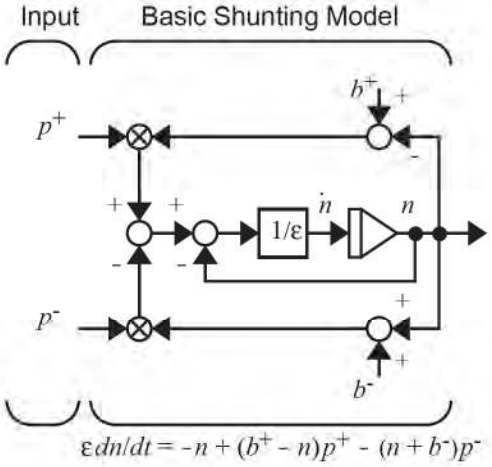


Figure P18.4 Shunting Network

The equation of operation of the shunting network is

$$\varepsilon\frac{dn(t)}{dt} = -n(t) + (b^+ - n(t))p^+ - (n(t) + b^-)p^-.$$

For the given parameter values this becomes

$$\frac{dn(t)}{dt} = -n(t) - (n(t) + 1)10 = -11n(t) - 10.$$

The solution to this equation is

$$n(t) = e^{-11t}n(0) + \int_0^t e^{-11(t-\tau)}(-10)d\tau,$$

or

$$n(t) = e^{-11t}0.5 + \left(-\frac{10}{11}\right)(1 - e^{-11t}).$$

The response is plotted in Figure P18.5.

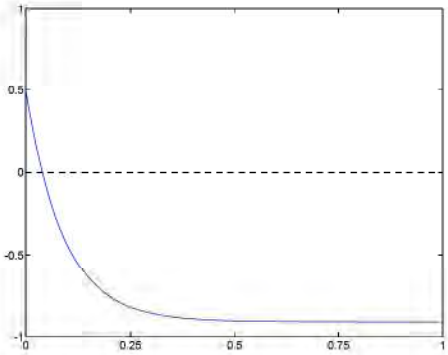


Figure P18.5 Shunting Network Response

There are two things to note about this response. First, as with all shunting networks, the response will never drop below $-b^-$, which in this case is -1. As the inhibitory input p^- is increased, the steady state response will decrease, but it can never be less than $-b^-$. The second characteristic of the response is that the speed of the response will increase as the input is increased. For instance, if the input were changed from $p^- = 10$ to $p^- = 100$, the response would be

$$n(t) = e^{-101t}0.5 + \left(-\frac{100}{101}\right)(1 - e^{-101t}).$$

Since e^{-101t} decays more rapidly than e^{-11t}, the response will be faster.

P18.4 Find the response of Layer 1 of the Grossberg network for the case of two neurons, with $^+b^1 = 1$, $^-b^1 = 0$, $\varepsilon = 1$ and input vector $\mathbf{p} = \begin{bmatrix} c & 2c \end{bmatrix}^T$. Assume that the initial conditions are set to zero. Demonstrate the effect of c on the response.

The Layer 1 differential equations for this case are

$$\frac{dn_1^1(t)}{dt} = -n_1^1(t) + (1 - n_1^1(t))(c) - n_1^1(t)(2c) = -(1 + 3c)n_1^1(t) + c,$$

$$\frac{dn_2^1(t)}{dt} = -n_2^1(t) + (1 - n_2^1(t))(2c) - n_2^1(t)(c) = -(1 + 3c)n_2^1(t) + 2c .$$

The solutions to these equations would be

$$n_1^1(t) = e^{-(1+3c)t} n_1^1(0) + \int_0^t e^{-(1+3c)(t-\tau)}(c)d\tau ,$$

$$n_2^1(t) = e^{-(1+3c)t} n_2^1(0) + \int_0^t e^{-(1+3c)(t-\tau)}(2c)d\tau .$$

If the initial conditions are set to zero, these equations reduce to

$$n_1^1(t) = \left(\frac{c}{1+3c}\right)(1 - e^{-(1+3c)t}) ,$$

$$n_2^1(t) = \left(\frac{2c}{1+3c}\right)(1 - e^{-(1+3c)t}) .$$

Note that the outputs of Layer 1 retain the same relative intensities as the inputs; the output of neuron 2 is always twice the output of neuron 1. This behavior is consistent with Eq. (18.13). In addition, the total output intensity ($n_1^1(t) + n_2^1(t)$) is never larger than ${}^+b^1 = 1$, as predicted in Eq. (18.14).

As c is increased, it has two effects on the response. First, the steady state values increase slightly. Second, the response becomes faster, since $e^{-(1+3c)t}$ decays more rapidly as c increases.

P18.5 Consider Layer 2 of the Grossberg network. Assume that the input to Layer 2 is applied for some length of time and then removed (set to zero).

i. Find a differential equation that describes the variation in the total output of Layer 2,

$$N^2(t) = \sum_{k=1}^{S^2} n_k^2(t) ,$$

after the input to Layer 2 has been removed.

ii. Find a differential equation that describes the variation in the relative outputs of Layer 2,

$$\bar{n}_i^2(t) = \frac{n_i^2(t)}{N^2(t)} ,$$

after the input to Layer 2 has been removed.

i. The operation of Layer 2 is described by Eq. (18.17):

$$\varepsilon\frac{d\mathbf{n}^2(t)}{dt} = -\mathbf{n}^2(t) + ({}^{+}\mathbf{b}^2 - \mathbf{n}^2(t))\{[{}^{+}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^2\mathbf{a}^1\} - (\mathbf{n}^2(t) + {}^{-}\mathbf{b}^2)[{}^{-}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) .$$

If the input is removed, then $\mathbf{W}^2\mathbf{a}^1$ is zero. For simplicity, we will set the inhibitory bias ${}^{-}\mathbf{b}^2$ to zero, and we will set all elements of the excitatory bias ${}^{+}\mathbf{b}^2$ to ${}^{+}b^2$. The response of neuron i is then given by

$$\varepsilon\frac{dn_i^2(t)}{dt} = -n_i^2(t) + ({}^{+}b^2 - n_i^2(t))\{f^2(n_i^2(t))\} - n_i^2(t)\left\{\sum_{k\neq i} f^2(n_k^2(t))\right\}.$$

This can be rearranged to produce

$$\varepsilon\frac{dn_i^2(t)}{dt} = -n_i^2(t) + {}^{+}b^2\{f^2(n_i^2(t))\} - n_i^2(t)\left\{\sum_{k=1}^{S^2} f^2(n_k^2(t))\right\}.$$

If we then make the definition

$$F^2(t) = \sum_{k=1}^{S^2} f^2(n_k^2(t)) ,$$

we can simplify the equation to

$$\varepsilon\frac{dn_i^2(t)}{dt} = -(1 + F^2(t))n_i^2(t) + {}^{+}b^2\{f^2(n_i^2(t))\} .$$

To get the total activity, sum this equation over i to produce

$$\varepsilon\frac{dN^2(t)}{dt} = -(1 + F^2(t))N^2(t) + {}^{+}b^2\{F^2(t)\} .$$

This equation describes the variation in the total activity in Layer 2 over time.

ii. The derivative of the relative activity is

$$\frac{d}{dt}[\bar{n}_i^2(t)] = \frac{d}{dt}\left[\frac{n_i^2(t)}{N^2(t)}\right] = \frac{1}{N^2(t)}\frac{d}{dt}[n_i^2(t)] - \left[\frac{n_i^2(t)}{(N^2(t))^2}\right]\frac{d}{dt}[N^2(t)].$$

If we then substitute our previous equations for these derivatives, we find

$$\varepsilon\frac{d}{dt}[\bar{n}_i^2(t)] = \frac{1}{N^2(t)}\left[\{-(1+F^2(t))n_i^2(t) + {}^+b^2\{f^2(n_i^2(t))\}\} - \frac{n_i^2(t)}{N^2(t)}\{-(1+F^2(t))N^2(t) + {}^+b^2\{F^2(t)\}\}\right].$$

Two terms on the right-hand side will cancel to produce

$$\varepsilon\frac{d}{dt}[\bar{n}_i^2(t)] = \frac{1}{N^2(t)}\left[\{{}^+b^2\{f^2(n_i^2(t))\}\} - \frac{n_i^2(t)}{N^2(t)}\{{}^+b^2\{F^2(t)\}\}\right],$$

or

$$\varepsilon\frac{d}{dt}[\bar{n}_i^2(t)] = \frac{{}^+b^2F^2(t)}{N^2(t)}\left[\frac{f^2(n_i^2(t))}{F^2(t)} - \frac{n_i^2(t)}{N^2(t)}\right].$$

We can put this in a more useful form if we expand the terms in the brackets:

$$\left[\frac{f^2(n_i^2(t))}{F^2(t)} - \frac{n_i^2(t)}{N^2(t)}\right] = \frac{1}{F^2(t)N^2(t)}[f^2(n_i^2(t))N^2(t) - n_i^2(t)F^2(t)]$$

$$= \frac{1}{F^2(t)N^2(t)}\left[g^2(n_i^2(t))n_i^2(t)\sum_{k=1}^{s^2} n_k^2(t) - n_i^2(t)\sum_{k=1}^{s^2} g^2(n_k^2(t))n_k^2(t)\right]$$

$$= \frac{n_i^2(t)}{F^2(t)N^2(t)}\left[\sum_{k=1}^{s^2} n_k^2(t)[g^2(n_i^2(t)) - g^2(n_k^2(t))]\right],$$

where

$$g^2(n_i^2(t)) = \frac{f^2(n_i^2(t))}{n_i^2(t)}.$$

Combining this expression with our previous equation, we obtain

$$\varepsilon \frac{d}{dt}[\bar{n}_i^2(t)] = {}^+b^2 \bar{n}_i^2(t) \left[\sum_{k=1}^{S^2} \bar{n}_k^2(t)[g^2(n_i^2(t)) - g^2(n_k^2(t))] \right].$$

This form of the differential equation describing the evolution of the relative outputs is very useful in demonstrating the characteristics of Layer 2, as we will see in the next solved problem.

P18.6 Suppose that the transfer function in Layer 2 of the Grossberg network is linear.

i. Show that the relative outputs of Layer 2 will not change after the input has been removed.

ii. Under what conditions will the total output of Layer 2 decay to zero after the input has been removed?

i. From Problem P18.5 we know that the relative outputs of Layer 2, after the input has been removed, evolve according to

$$\varepsilon \frac{d}{dt}[\bar{n}_i^2(t)] = {}^+b^2 \bar{n}_i^2(t) \left[\sum_{k=1}^{S^2} \bar{n}_k^2(t)[g^2(n_i^2(t)) - g^2(n_k^2(t))] \right].$$

If the transfer function for Layer 2, $f^2(n)$, is linear, then

$$f^2(n) = c\, n.$$

Therefore

$$g^2(n) = \frac{f^2(n)}{n} = \frac{c\, n}{n} = c.$$

If we substitute this expression into our differential equation, we find

$$\varepsilon \frac{d}{dt}[\bar{n}_i^2(t)] = {}^+b^2 \bar{n}_i^2(t) \left[\sum_{k=1}^{S^2} \bar{n}_k^2(t)[c - c] \right] = 0.$$

Therefore the relative outputs do not change.

ii. From Problem P18.5, the total output of Layer 2, after the input has been removed, evolves according to

$$\varepsilon \frac{dN^2(t)}{dt} = -(1 + F^2(t))N^2(t) + {}^+b^2\{F^2(t)\}.$$

If $f^2(n)$ is linear, then

$$F^2(t) = \sum_{k=1}^{S^2} f^2(n_k^2(t)) = \sum_{k=1}^{S^2} c\, n_k^2(t) = c \sum_{k=1}^{S^2} n_k^2(t) = c\, N^2(t).$$

Therefore the differential equation can be written

$$\varepsilon \frac{dN^2(t)}{dt} = -(1 + c\, N^2(t))N^2(t) + {}^+b^2\{c\, N^2(t)\} = -\{1 - {}^+b^2 c + c\, N^2(t)\}N^2(t).$$

To find the equilibrium solutions of this equation, we set the derivative to zero:

$$0 = -\{1 - {}^+b^2 c + c\, N^2(t)\}N^2(t).$$

Therefore there are two equilibrium solutions:

$$N^2(t) = 0 \text{ or } N^2(t) = \frac{{}^+b^2 c - 1}{c}.$$

We want to know the conditions under which the total output will converge to each of these possible solutions. Consider two cases:

1. $1 \geq {}^+b^2 c$

   For this case, the derivative of the total output,

   $$\varepsilon \frac{dN^2(t)}{dt} = -\{1 - {}^+b^2 c + c\, N^2(t)\}N^2(t),$$

   will always be negative for positive $N^2(t)$. (Recall that the outputs of Layer 2 are never negative.) Therefore, the total output will decay to zero.

   $$\lim_{t \to \infty} N^2(t) = 0$$

2. $1 < {}^+b^2 c$

   (a) If $N^2(0) > ({}^+b^2 c - 1)/c$, then the derivative of the total output will be negative until $N^2(t) = ({}^+b^2 c - 1)/c$, when the derivative will be zero. Therefore,

   $$\lim_{t \to \infty} N^2(t) = \frac{({}^+b^2 c - 1)}{c}.$$

(b) If $N^2(0) < ({}^+b^2 c - 1)/c$, then the derivative of the total output will be positive until $N^2(t) = ({}^+b^2 c - 1)/c$, when the derivative will be zero. Therefore,

$$\lim_{t \to \infty} N^2(t) = \frac{({}^+b^2 c - 1)}{c}.$$

Therefore, if the transfer function of Layer 2 is linear, the total output will decay to zero if $1 \geq {}^+b^2 c$. If $1 < {}^+b^2 c$, then the total output will converge to $({}^+b^2 c - 1)/c$. In any case, the relative outputs will remain constant.

As an example of these results, consider the following Layer 2 equations:

$$\frac{dn_1^2(t)}{dt} = -n_1^2(t) + (1.5 - n_1^2(t))\{n_1^2(t)\} - n_1^2(t)\{n_2^2(t)\},$$

$$\frac{dn_2^2(t)}{dt} = -n_2^2(t) + (1.5 - n_2^2(t))\{n_2^2(t)\} - n_2^2(t)\{n_1^2(t)\}.$$

For this case, $\varepsilon = 1$, ${}^+b^2 = 1.5$ and $c = 1$, therefore $1 < {}^+b^2 c$. The total output will converge to

$$\lim_{t \to \infty} N^2(t) = \frac{({}^+b^2 c - 1)}{c} = \frac{(1.5 - 1)}{1} = 0.5.$$

In Figure P18.6 we can see the response of Layer 2 for two different sets of initial conditions:

$$\mathbf{n}^2(0) = \begin{bmatrix} 0.75 \\ 0.5 \end{bmatrix} \text{ and } \mathbf{n}^2(0) = \begin{bmatrix} 0.15 \\ 0.1 \end{bmatrix}.$$

As expected, the total output converges to 0.5 for both initial conditions. In addition, since the relative values of the initial conditions are the same for the two cases, the outputs converge to the same values in both cases.

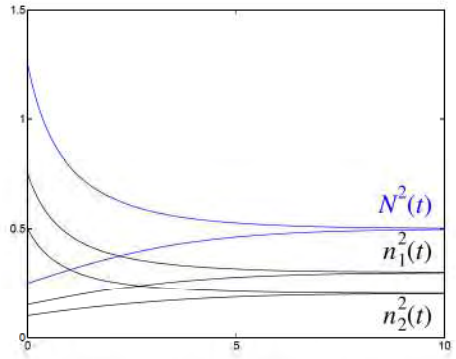


Figure P18.6 Response of Layer 2 for Linear $f^2(n)$

P18.7 Show that the continuous-time Hebb rule with decay, given by Eq. (18.24), is equivalent to the discrete-time version given by Eq. (15.18).

The continuous-time Hebb rule with decay is

$$\frac{dw_{i,j}^2(t)}{dt} = \alpha\{-w_{i,j}^2(t) + n_i^2(t)n_j^1(t)\}.$$

If we approximate the derivative by

$$\frac{dw_{i,j}^2(t)}{dt} \approx \frac{w_{i,j}^2(t+\Delta t) - w_{i,j}^2(t)}{\Delta t},$$

the Hebb rule becomes

$$w_{i,j}^2(t+\Delta t) = w_{i,j}^2(t) + \alpha\Delta t\{-w_{i,j}^2(t) + n_i^2(t)n_j^1(t)\}.$$

This can be rearranged to obtain

$$w_{i,j}^2(t+\Delta t) = [1-\alpha\Delta t]w_{i,j}^2(t) + \alpha\Delta t\{n_i^2(t)n_j^1(t)\}.$$

In vector form this would be

$$\mathbf{W}^2(t+\Delta t) = [1-\alpha\Delta t]\mathbf{W}^2(t) + \alpha\Delta t\{\mathbf{n}^2(t)(\mathbf{n}^1(t))^T\}.$$

If we compare this with Eq. (15.18),

$$\mathbf{W}(q) = (1-\gamma)\mathbf{W}(q-1) + \alpha\mathbf{a}(q)\mathbf{p}^T(q),$$

we can see that they have the identical form.

Epilogue

The Grossberg network presented in this chapter was inspired by the visual system of higher vertebrates. To motivate the network, we presented a brief description of the primary visual pathway. We also discussed some visual illusions, which help us to understand the mechanisms underlying the visual system.

The Grossberg network is a two-layer, continuous-time competitive network, which is very similar in structure and operation to the Kohonen competitive network presented in Chapter 14. The first layer of the Grossberg network normalizes the input pattern. It demonstrates how the visual system can use on-center/off-surround connection patterns and a shunting model to implement an automatic gain control, which normalizes total activity.

The second layer of the Grossberg network performs a competition, which contrast enhances the output pattern and stores it in short-term memory. It uses nonlinear feedback and the on-center/off-surround connection pattern to produce the competition and the storage. The choice of the transfer function and the feedback connection pattern determines the degree of competition (e.g., winner-take-all, mild contrast enhancement, or no change in the pattern).

The adaptive weights in the Grossberg network use an instar learning rule, which stores prototype patterns in long-term memory. When a winner-take-all competition is performed in the second layer, this learning rule is equivalent to the Kohonen learning rule used in Chapter 14.

As with the Kohonen network, one key problem of the Grossberg network is the stability of learning; as more inputs are applied to the network, the weight matrix may never converge. This problem was discussed extensively in Chapter 14. In Chapter 16 we will present a class of networks that is designed to overcome this difficulty: the Adaptive Resonance Theory (ART) networks. The ART networks are direct descendents of the Grossberg network presented in this chapter.

Another problem with the Grossberg network, which we have not discussed in this chapter, is the stability of the differential equations that implement the network. In Layer 2, for example, we have a set of differential equations with nonlinear feedback. Can we make some general statement about the stability of such systems? Chapter 17 will present a comprehensive discussion of this problem.

Further Reading

[GrMi89] S. Grossberg, E. Mingolla and D. Todorovic, "A neural network architecture for preattentive vision," *IEEE Transactions on Biomedical Engineering*, vol. 36, no. 1, pp. 65–84, 1989.

The objective of this paper is to develop a neural network for general purpose preattentive vision. The network consists of two main subsystems: a boundary contour system and a feature contour system.

[Gros76] S. Grossberg, "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors," *Biological Cybernetics*, vol. 23, pp. 121–134, 1976.

Grossberg describes a continuous-time competitive network, inspired by the developmental physiology of the visual cortex. The structure of this network forms the foundation for other important networks.

[Gros82] S. Grossberg, *Studies of Mind and Brain*, Boston: D. Reidel Publishing Co., 1982.

This book is a collection of Stephen Grossberg papers from the period 1968 through 1980. It covers many of the fundamental concepts that are used in later Grossberg networks, such as the adaptive resonance theory networks.

[Hube88] D.H. Hubel, *Eye, Brain, and Vision*, New York: Scientific American Library, 1988.

David Hubel has been at the center of research in this area for 30 years, and his book provides an excellent introduction to the human visual system. He explains the current view of the visual system in a way that is easily accessible to anyone with some scientific training.

[vanT75] H. F. J. M. van Tuijl, "A new visual illusion: Neonlike color spreading and complementary color induction between subjective contours," *Acta Psychologica*, vol. 39, pp. 441–445, 1975.

This paper describes the original discovery of the illusion in which crosses of certain colors, when placed inside Ehrenstein figures, appear to spread into solid shapes.

[vond73] C. von der Malsburg, "Self-organization of orientation sensitive cells in the striate cortex," *Kybernetic*, vol. 14, pp. 85–100, 1973.

Malsberg's is one of the first papers to present a self-organizing feature map neural network. The network is a model for the visual cortex of higher vertebrates. This paper influenced the work of Kohonen and Grossberg on feature maps.

Exercises

E18.1 Consider the leaky integrator shown in Figure E18.1.

i. Find the response $n(t)$ if $\varepsilon = 1$, $n(0) = 1$ and $p(t) = 0.5$.

ii. Find the response $n(t)$ if $\varepsilon = 1$, $n(0) = 1$ and $p(t) = 2$.

iii. Find the response $n(t)$ if $\varepsilon = 4$, $n(0) = 1$ and $p(t) = 2$.



iv. Check your answers to the previous parts by writing a MATLAB M-file to simulate the leaky integrator. Use the **ode45** routine. Plot the response for each case.

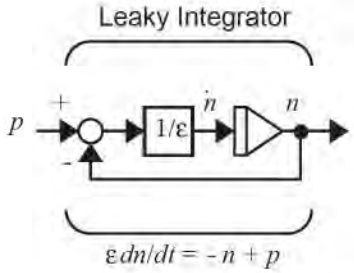


Figure E18.1 Leaky Integrator

E18.2 Consider the shunting network shown in Figure E18.2.

i. Find and sketch the response of the shunting network if $\varepsilon = 2$, $b^+ = 3$, $b^- = 1$, $p^+ = 0$, $p^- = 5$ and $n(0) = 1$.

ii. Find and sketch the response of the shunting network if $\varepsilon = 2$, $b^+ = 3$, $b^- = 1$, $p^+ = 0$, $p^- = 50$ and $n(0) = 1$.

iii. Find and sketch the response of the shunting network if $\varepsilon = 2$, $b^+ = 3$, $b^- = 1$, $p^+ = 50$, $p^- = 0$ and $n(0) = 1$.

iv. Find and sketch the response of the shunting network if $\varepsilon = 5$, $b^+ = 2$, $b^- = 6$, $p^+ = 5$, $p^- = 0$ and $n(0) = 0$.

v. Find and sketch the response of the shunting network if $\varepsilon = 5$, $b^+ = 2$, $b^- = 6$, $p^+ = 0$, $p^- = 5$ and $n(0) = 0$.

vi. Find and sketch the response of the shunting network if $\varepsilon = 0.25$, $b^+ = 4$, $b^- = 2$, $p^+ = 2$, $p^- = 2$ and $n(0) = 0$.

vii. Find and sketch the response of the shunting network if $\varepsilon = 0.25$, $b^+ = 4$, $b^- = 2$, $p^+ = 2$, $p^- = 4$ and $n(0) = 0$.

```
» 2 + 2
ans =
    4
```

viii. Check your answers to the previous parts by writing a MATLAB M-file to simulate the shunting network. Use the **ode45** routine. Plot the response for each case. Verify that your responses agree with the known characteristics of the shunting model.

ix. Explain the differences in operation of the leaky integrator and the shunting network.

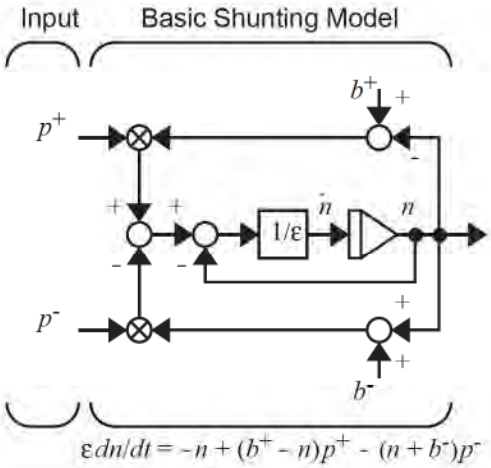


Figure E18.2 Shunting Network

E18.3 Suppose that Layer 1 of the Grossberg network has two neurons, with ${}^+b^1 = 0.5$, $\varepsilon = 0.5$ and input vector $\mathbf{p} = \begin{bmatrix} 2 & 1 \end{bmatrix}^T$. Assume that the initial conditions are set to zero.

i. Find the steady state response of Layer 1, using Eq. (18.13).

ii. Find the solution to the differential equation for Layer 1. Verify that the steady state response agrees with your answer to part (i).

```
» 2 + 2
ans =
    4
```

iii. Check your answer by writing a MATLAB M-file to simulate Layer 1 of the Grossberg network. Use the **ode45** routine. Plot the response.

E18.4 Repeat Exercise E18.3 for input vector $\mathbf{p} = \begin{bmatrix} 20 & 10 \end{bmatrix}^T$.

E18.5 Consider the first layer of the Grossberg network. The parameters are set to be ${}^+b^1 = 2$, ${}^-b^1 = 0$, $\varepsilon = 2$. The input to the network is $\mathbf{p} = \begin{bmatrix} 4 & 1 \end{bmatrix}^T$. Find the first layer outputs and sketch them versus time.

E18.6 Find the differential equation that describes the variation in the total output of Layer 1,

$$N^1(t) = \sum_{i=1}^{S^1} n_i^1(t).$$

(Use the technique presented in Problem P18.5.)

E18.7 Assume that Layer 2 of the Grossberg network has two neurons, with $f^2(n) = 2n$, $\varepsilon = 1$, ${}^+b^2 = 1$ and ${}^-b^2 = 0$. The inputs have been applied for some length of time, then removed.

i. What will be the steady state total output, $\lim_{t \to \infty} N^2(t)$?

ii. Repeat part (i) if ${}^+b^2 = 0.25$.



iii. Check your answers to the previous parts by writing a MATLAB M-file to simulate Layer 2 of the Grossberg network. Use the **ode45** routine. Plot the responses for the following initial conditions:

$$\mathbf{n}^2(0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \text{ and } \mathbf{n}^2(0) = \begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}.$$

E18.8 Suppose that the transfer function for Layer 2 of the Grossberg network is $f^2(n) = c \times (n)^2$, and $\varepsilon = 1$, ${}^+b^2 = 1$.

i. Using the results of Problem P18.5, show that, after the inputs have been removed, all of the relative outputs of Layer 2 will decay to zero, except the one with the largest initial condition (winner-take-all competition).

ii. For what values of c will the total output $N^2(t)$ have a nonzero stable point (steady state value)?

iii. If the condition of part (ii) is satisfied, what will be the steady state value of $N^2(t)$? Will this depend on the initial condition $N^2(0)$?



iv. Check your answers to the previous parts by writing a MATLAB M-file and simulating the total response of Layer 2 for $c = 4$ and $N^2(0) = 3$.

E18.9 Simulate the response of the adaptive weights for the Grossberg network. Assume that the coefficient ε is 1. Assume that two different input patterns are alternately presented to the network for periods of 0.2 seconds at a time. Also, assume that Layer 1 and Layer 2 converge very quickly, in comparison with the convergence of the weights, so that the neuron out-



puts are effectively constant over the 0.2 seconds. The Layer 2 and Layer 1 outputs for the two different input patterns will be:

$$\text{for pattern 1: } \mathbf{n}^1 = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}, \ \mathbf{n}^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$\text{for pattern 2: } \mathbf{n}^1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \ \mathbf{n}^2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

E18.10 Repeat Exercise E18.9, but use the Hebb rule with decay, Eq. (18.24), instead of the instar learning of Eq. (18.25). Explain the differences between the two responses.



19 Adaptive Resonance Theory

Objectives	19-1
Theory and Examples	19-2
Overview of Adaptive Resonance	19-2
Layer 1	19-4
Steady State Analysis	19-6
Layer 2	19-10
Orienting Subsystem	19-13
Learning Law: L1-L2	19-16
Subset/Superset Dilemma	19-17
Learning Law	19-18
Learning Law: L2-L1	19-20
ART1 Algorithm Summary	19-21
Initialization	19-21
Algorithm	19-21
Other ART Architectures	19-23
Summary of Results	19-24
Solved Problems	19-29
Epilogue	19-44
Further Reading	19-45
Exercises	19-47

Objectives

In Chapter 16 and Chapter 18 we learned that one key problem of competitive networks is the stability of learning. There is no guarantee that, as more inputs are applied to the network, the weight matrix will eventually converge. In this chapter we will present a modified type of competitive learning, called adaptive resonance theory (ART), which is designed to overcome the problem of learning stability.

Theory and Examples

A key problem of the Grossberg network presented in Chapter 18 and the competitive networks of Chapter 16, is that they do not always form stable clusters (or categories). Grossberg did show [Gros76] that if the number of input patterns is not too large, or if the input patterns do not form too many clusters relative to the number of neurons in Layer 2, then the learning eventually stabilizes. However, he also showed that the standard competitive networks do not have stable learning in response to arbitrary input patterns. The learning instability occurs because of the network's adaptability (or plasticity), which causes prior learning to be eroded by more recent learning.

Stability/Plasticity

Grossberg refers to this problem as the "*stability / plasticity dilemma*." How can a system be receptive to significant new patterns and yet remain stable in response to irrelevant patterns? We know that biological systems are very good at this. For example, you can easily recognize your mother's face, even if you have not seen her for a long time and have met many new people in the mean time.

Grossberg and Gail Carpenter developed a theory, called adaptive resonance theory (ART), to address the stability/plasticity dilemma (see [CaGr87a], [CaGr87b], [CaGr90], [CaGrRe91] and [CaGrMa92]). The ART networks are based on the Grossberg network of Chapter 18. The key innovation of ART is the use of "expectations." As each input pattern is presented to the network, it is compared with the prototype vector that it most closely matches (the expectation). If the match between the prototype and the input vector is not adequate, a new prototype is selected. In this way, previously learned memories (prototypes) are not eroded by new learning.

It is beyond the scope of this text to discuss all of the variations of adaptive resonance theory. Instead, we will present one of the ART networks in detail — ART1 (see [CaGr87a]). This particular network is designed for binary input vectors only. However, from this one architecture, the key features of adaptive resonance theory can be understood.

Overview of Adaptive Resonance

The basic ART architecture is shown in Figure 19.1. It is a modification of the Grossberg network of Chapter 18 (compare with Figure 18.13), which is designed to stabilize the learning process. The innovations of the ART architecture consist of three parts: Layer 2 (L2) to Layer 1 (L1) expectations, the orienting subsystem and gain control. In this section we will describe the general operation of the ART system; then, in later sections, we will discuss each subsystem in detail.

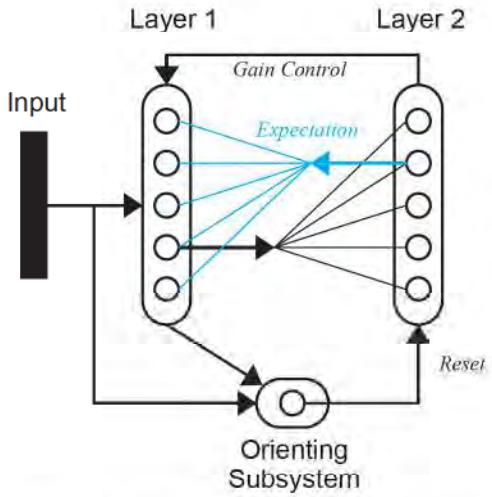


Figure 19.1 Basic ART Architecture

Recall from Chapter 18 that the L1-L2 connections of the Grossberg network are instars, which perform a clustering (or categorization) operation. When an input pattern is presented to the network, it is multiplied (after normalization) by the L1-L2 weight matrix. Then, a competition is performed at Layer 2 to determine which row of the weight matrix is closest to the input vector. That row is then moved toward the input vector. After learning is complete, each row of the L1-L2 weight matrix is a prototype pattern, which represents a cluster (or category) of input vectors.

In the ART networks, learning also occurs in a set of feedback connections from Layer 2 to Layer 1. These connections are outstars (see Chapter 15), which perform pattern recall. When a node in Layer 2 is activated, this reproduces a prototype pattern (the expectation) at Layer 1. Layer 1 then performs a comparison between the expectation and the input pattern.

When the expectation and the input pattern are not closely matched, the orienting subsystem causes a reset in Layer 2. This reset disables the current winning neuron, and the current expectation is removed. A new competition is then performed in Layer 2, while the previous winning neuron is disabled. The new winning neuron in Layer 2 projects a new expectation to Layer 1, through the L2-L1 connections. This process continues until the L2-L1 expectation provides a close enough match to the input pattern.

In the following sections we will investigate each of the subsystems of the ART system, as they apply to one particular ART network — ART1 ([CaGr87a]). We will first describe the differential equations that describe the subsystem operations. Then we will derive the steady state responses of each subsystem. Finally, we will summarize the overall operation of the ART1 system.

Layer 1

The main purpose of Layer 1 is to compare the input pattern with the expectation pattern from Layer 2. (*Both patterns are binary in ART1.*) If the patterns are not closely matched, the orienting subsystem will cause a reset in Layer 2. If the patterns are close enough, Layer 1 combines the expectation and the input to form a new prototype pattern.

Layer 1 of the ART1 network, which is displayed in Figure 19.2, is very similar to Layer 1 of the Grossberg network (see Figure 18.14). The differences occur at the excitatory and inhibitory inputs to the shunting model. For the ART1 network, no normalization is performed at Layer 1; therefore we don't have the on-center/off-surround connections from the input vector. The excitatory input to Layer 1 of ART1 consists of a combination of the input pattern and the L1-L2 expectation. The inhibitory input consists of the gain control signal from Layer 2. In the following we will explain how these inputs work together.

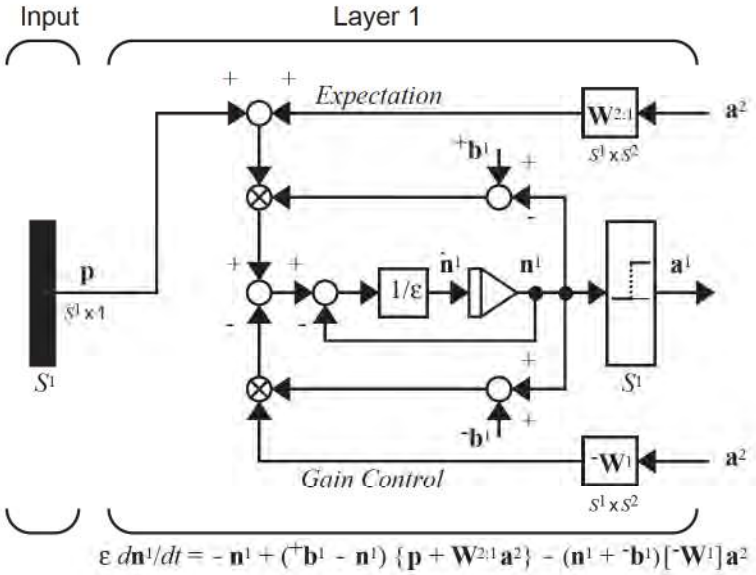


Figure 19.2 Layer 1 of the ART1 Network

The equation of operation of Layer 1 is

$$\varepsilon \frac{d\mathbf{n}^1(t)}{dt} = -\mathbf{n}^1(t) + (^+\mathbf{b}^1 - \mathbf{n}^1(t))\{\mathbf{p} + \mathbf{W}^{2:1}\mathbf{a}^2(t)\} - (\mathbf{n}^1(t) + {}^-\mathbf{b}^1)[{}^-\mathbf{W}^1]\mathbf{a}^2(t) \tag{19.1}$$

and the output of Layer 1 is computed

$$\mathbf{a}^1 = \mathbf{hardlim}^+(\mathbf{n}^1), \tag{19.2}$$

where

$$hardlim^+(n) = \begin{cases} 1, & n > 0 \\ 0, & n \le 0 \end{cases}. \tag{19.3}$$

Eq. (19.1) is a shunting model with excitatory input $\mathbf{p} + \mathbf{W}^{2:1}\mathbf{a}^2(t)$, which is the sum of the input vector and the L2-L1 expectation. For example, assume that the jth neuron in Layer 2 has won the competition, so that its output is 1, and the other neurons have zero output. For this case we have

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} \mathbf{w}_1^{2:1} & \mathbf{w}_2^{2:1} & \cdots & \mathbf{w}_j^{2:1} & \cdots & \mathbf{w}_{S^2}^{2:1} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} = \mathbf{w}_j^{2:1}, \tag{19.4}$$

where $\mathbf{w}_j^{2:1}$ is the jth column of the matrix $\mathbf{W}^{2:1}$. (The $\mathbf{W}^{2:1}$ matrix is trained using an outstar rule, as we will show in a later section.) Now we can see that

$$\mathbf{p} + \mathbf{W}^{2:1}\mathbf{a}^2 = \mathbf{p} + \mathbf{w}_j^{2:1}. \tag{19.5}$$

Therefore the excitatory input to Layer 1 is the sum of the input pattern and the L2-L1 expectation. Each column of the L2-L1 matrix represents a different expectation (prototype pattern). Layer 1 combines the input pattern with the expectation using an AND operation, as we will see later.

The inhibitory input to Layer 1 is the gain control term $[\bar{\mathbf{W}}^1]\mathbf{a}^2(t)$, where

$$\bar{\mathbf{W}}^1 = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}. \tag{19.6}$$

Therefore, the inhibitory input to each neuron in Layer 1 is the sum of all of the outputs of Layer 2. Since we will be using a winner-take-all competition in Layer 2, whenever Layer 2 is active there will be one, and only one, nonzero element of $\mathbf{a}^2$ after the competition. Therefore the gain control input to Layer 1 will be one when Layer 2 is active, and zero when Layer 2 is inactive (all neurons having zero output). The purpose of this gain control will become apparent as we analyze the steady state behavior of Layer 1.

Steady State Analysis

The response of neuron i in Layer 1 is described by

$$\varepsilon \frac{dn_i^1}{dt} = -n_i^1 + ({}^+b^1 - n_i^1)\left\{p_i + \sum_{j=1}^{S^2} w_{i,j}^{2:1} a_j^2\right\} - (n_i^1 + {}^-b^1)\sum_{j=1}^{S^2} a_j^2, \qquad (19.7)$$

where $\varepsilon \ll 1$ so that the short-term memory traces (the neuron outputs) change much faster than the long-term memory traces (the weight matrices).

We want to investigate the steady state response of this system for two different cases. In the first case Layer 2 is inactive, therefore $a_j^2 = 0$ for all j. In the second case Layer 2 is active, and therefore one neuron has an output of 1, and all other neurons output 0.

Consider first the case where Layer 2 is inactive. Since each $a_j^2 = 0$, Eq. (19.7) simplifies to

$$\varepsilon \frac{dn_i^1}{dt} = -n_i^1 + ({}^+b^1 - n_i^1)\{p_i\}. \qquad (19.8)$$

In the steady state ($dn_i^1(t)/dt = 0$) we have

$$0 = -n_i^1 + ({}^+b^1 - n_i^1)p_i = -(1 + p_i)n_i^1 + {}^+b^1 p_i. \qquad (19.9)$$

If we solve for the steady state neuron output n_i^1 we find

$$n_i^1 = \frac{{}^+b^1 p_i}{1 + p_i}. \qquad (19.10)$$

Therefore, if $p_i = 0$ then $n_i^1 = 0$, and if $p_i = 1$ then $n_i^1 = {}^+b^1/2 > 0$. Since we chose the transfer function for Layer 1 to be the $hardlim^+$ function, then we have

$$\mathbf{a}^1 = \mathbf{p}. \qquad (19.11)$$

Therefore, when Layer 2 is inactive, the output of Layer 1 is the same as the input pattern.

Now let's consider the second case, where Layer 2 is active. Assume that neuron j is the winning neuron in Layer 2. Then $a_j^2 = 1$ and $a_k^2 = 0$ for $k \neq j$. For this case Eq. (19.7) simplifies to

$$\varepsilon \frac{dn_i^1}{dt} = -n_i^1 + ({}^+b^1 - n_i^1)\{p_i + w_{i,j}^{2:1}\} - (n_i^1 + {}^-b^1). \tag{19.12}$$

In the steady state ($dn_i^1(t)/dt = 0$) we have

$$\begin{aligned} 0 &= -n_i^1 + ({}^+b^1 - n_i^1)\{p_i + w_{i,j}^{2:1}\} - (n_i^1 + {}^-b^1) \\ &= -(1 + p_i + w_{i,j}^{2:1} + 1)n_i^1 + ({}^+b^1(p_i + w_{i,j}^{2:1}) - {}^-b^1). \end{aligned} \tag{19.13}$$

If we solve for the steady state neuron output n_i^1 we find

$$n_i^1 = \frac{{}^+b^1(p_i + w_{i,j}^{2:1}) - {}^-b^1}{2 + p_i + w_{i,j}^{2:1}}. \tag{19.14}$$

Recall that Layer 1 should combine the input vector with the expectation from Layer 2 (represented by $\mathbf{w}_j^{2:1}$). Since we are dealing with binary patterns (both the input and the expectation), we will use a logical AND operation to combine the two vectors. In other words, we want n_i^1 to be less than zero when either p_i or $w_{i,j}^{2:1}$ is equal to zero, and we want n_i^1 to be greater than zero when both p_i and $w_{i,j}^{2:1}$ are equal to one.

If we apply these conditions to Eq. (19.14), we obtain the following equations:

$${}^+b^1(2) - {}^-b^1 > 0, \tag{19.15}$$

$${}^+b^1 - {}^-b^1 < 0, \tag{19.16}$$

which we can combine to produce

$${}^+b^1(2) > {}^-b^1 > {}^+b^1. \tag{19.17}$$

For example, we can use ${}^+b^1 = 1$ and ${}^-b^1 = 1.5$ to satisfy these conditions.

Therefore, if Eq. (19.17) is satisfied, and neuron j of Layer 2 is active, then the output of Layer 1 will be

$$\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1}, \tag{19.18}$$

where $\cap$ represents the logical AND operation.

Notice that we needed the gain control in order to implement the AND operation. Consider the numerator of Eq. (19.14):

$${}^+b^1(p_i + w_{i,j}^{2:1}) - {}^-b^1. \tag{19.19}$$

The term ${}^{-}b^{1}$ is multiplied by the gain control term, which in this case is 1. If this term did not exist, then Eq. (19.19) would be greater than zero (and therefore n_i^1 would be greater than zero) whenever either p_i or $w_{i,j}^{2:1}$ was greater than zero. This would represent an OR operation, rather than an AND operation. As we will see when we discuss the orienting subsystem, it is critical that Layer 1 perform an AND operation.

When Layer 2 is inactive, the gain control term is zero. This is necessary because in that case we want Layer 1 to respond to the input pattern alone, since no expectation will be activated by Layer 2.

To summarize the steady state operation of Layer 1:

If Layer 2 is not active (i.e., each $a_j^2 = 0$),

$$\mathbf{a}^1 = \mathbf{p}. \tag{19.20}$$

If Layer 2 is active (i.e., one $a_j^2 = 1$),

$$\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1}. \tag{19.21}$$



To demonstrate the operation of Layer 1, assume the following network parameters:

$$\varepsilon = 0.1, \quad {}^{+}b^{1} = 1 \text{ and } {}^{-}b^{1} = 1.5. \tag{19.22}$$

Assume also that we have two neurons in Layer 2, two elements in the input vector and the following weight matrix and input:

$$\mathbf{W}^{2:1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } \mathbf{p} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{19.23}$$

If we take the case where Layer 2 is active, and neuron 2 of Layer 2 wins the competition, the equations of operation of Layer 1 are

$$(0.1)\frac{dn_1^1}{dt} = -n_1^1 + (1 - n_1^1)\{p_1 + w_{1,2}^{2:1}\} - (n_1^1 + 1.5) \tag{19.24}$$

$$= -n_1^1 + (1 - n_1^1)\{0 + 1\} - (n_1^1 + 1.5) = -3n_1^1 - 0.5$$

$$(0.1)\frac{dn_2^1}{dt} = -n_2^1 + (1 - n_2^1)\{p_2 + w_{2,2}^{2:1}\} - (n_2^1 + 1.5) \tag{19.25}$$

$$= -n_2^1 + (1 - n_2^1)\{1 + 1\} - (n_2^1 + 1.5) = -4n_2^1 + 0.5.$$

These can be simplified to obtain

$$\frac{dn_1^1}{dt} = -30n_1^1 - 5\,, \tag{19.26}$$

$$\frac{dn_2^1}{dt} = -40n_2^1 + 5\,. \tag{19.27}$$

In this simple case we can find closed-form solutions for these equations. If we assume that both neurons start with zero initial conditions, the solutions are

$$n_1^1(t) = -\frac{1}{6}[1 - e^{-30t}]\,, \tag{19.28}$$

$$n_2^1(t) = \frac{1}{8}[1 - e^{-40t}]\,. \tag{19.29}$$

These are displayed in Figure 19.3.

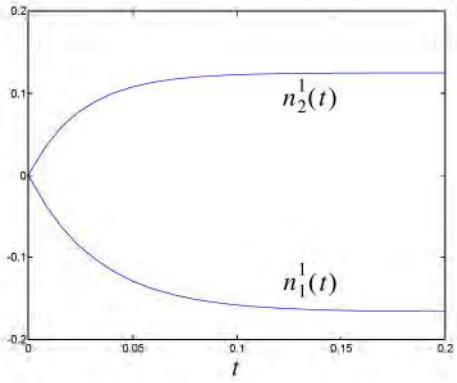


Figure 19.3 Response of Layer 1

Note that $n_1^1(t)$ converges to a negative value, and $n_2^1(t)$ converges to a positive value. Therefore, $a_1^1(t)$ converges to 0, and $a_2^1(t)$ converges to 1 (recall that the transfer function for Layer 1 is $hardlim^+$). This agrees with our steady state analysis (see Eq. (19.21)), since

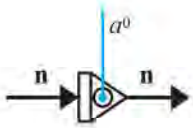
$$\mathbf{p} \cap \mathbf{w}_2^{2:1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cap \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \mathbf{a}^1\,. \tag{19.30}$$



To experiment with Layer 1 of the ART1 network, use the Neural Network Design Demonstration ART1 Layer 1 (`nnd16al1`).

Layer 2

Layer 2 of the ART1 network is almost identical to Layer 2 of the Grossberg network of Chapter 18. Its main purpose is to contrast enhance its output pattern. For our implementation of the ART1 network, the contrast enhancement will be a winner-take-all competition, so only the neuron that receives the largest input will have a nonzero output.



There is one major difference between the second layers of the Grossberg and the ART1 networks. Layer 2 of the ART1 network uses an integrator that can be reset. In this type of integrator, whose symbol is shown in the left margin, any positive outputs are reset to zero whenever the a^0 signal becomes positive. The outputs that are reset remain inhibited for a long period of time, so that they cannot be driven above zero. (By a "long" period of time we mean until an adequate match has occurred and the weights have been updated.)

In the original ART1 paper, Carpenter and Grossberg suggested that the reset mechanism could be implemented using a gated dipole field [CaGr87]. They later suggested a more sophisticated biological model, using chemical neurotransmitters, in their ART3 architecture [CaGr90]. For our purposes we will not be concerned with the specific biological implementation.

Figure 19.4 displays the complete Layer 2 of the ART1 network. Again, it is almost identical to Layer 2 of the Grossberg network (see Figure 18.16), with the primary exception of the resetable integrator. The reset signal, a^0, is the output of the orienting subsystem, which we will discuss in the next section. It generates a reset whenever there is a mismatch at Layer 1 between the input signal and the L2-L1 expectation.

One other small difference between Layer 2 of the ART1 network and Layer 2 of the Grossberg network is that two transfer functions are used in ART1. The transfer function $\mathbf{f}^2(\mathbf{n}^2)$ is used for the on-center/off-surround feedback connections, while the output of Layer 2 is computed as $\mathbf{a}^2 = \mathbf{hardlim}^+(\mathbf{n}^2)$. The reason for the second transfer function is that we want the output of Layer 2 to be a binary signal.

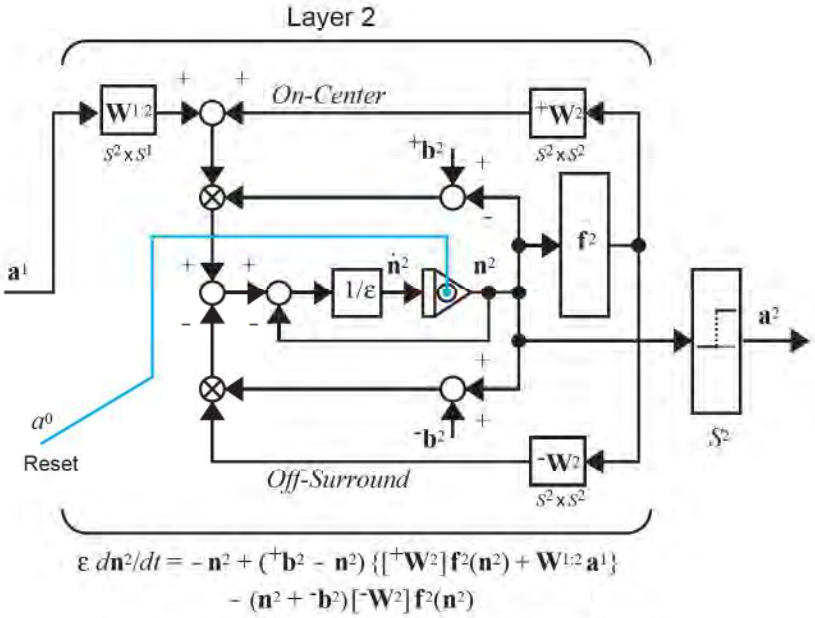


Figure 19.4 Layer 2 of the ART1 Network

The equation of operation of Layer 2 is

$$\varepsilon \frac{d\mathbf{n}^2(t)}{dt} = -\mathbf{n}^2(t) + (^+\mathbf{b}^2 - \mathbf{n}^2(t))\{[^+\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^{1:2}\mathbf{a}^1\} \qquad (19.31)$$

$$- (\mathbf{n}^2(t) + {}^-\mathbf{b}^2)[^-\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) .$$

This is a shunting model with excitatory input $\{[^+\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^{1:2}\mathbf{a}^1\}$, where $^+\mathbf{W}^2$ provides on-center feedback connections (identical to Layers 1 and 2 of the Grossberg network of Chapter 18, Eq. (18.6)), and $\mathbf{W}^{1:2}$ consists of adaptive weights, analogous to the weights in the Kohonen network. They are trained according to an instar rule, as we will see in a later section. The rows of $\mathbf{W}^{1:2}$, after training, will represent the prototype patterns.

The inhibitory input to the shunting model is $[^-\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t))$, where $^-\mathbf{W}^2$ provides off-surround feedback connections (identical to Layers 1 and 2 of the Grossberg network — Eq. (18.7)).



To illustrate the performance of Layer 2, consider a two-neuron layer with

$$\varepsilon = 0.1\,,\ {}^{+}\mathbf{b}^2 = \begin{bmatrix}1\\1\end{bmatrix},\ {}^{-}\mathbf{b}^2 = \begin{bmatrix}1\\1\end{bmatrix},\ \mathbf{W}^{1:2} = \begin{bmatrix}({}_1\mathbf{w}^{1:2})^T\\({}_2\mathbf{w}^{1:2})^T\end{bmatrix} = \begin{bmatrix}0.5 & 0.5\\1 & 0\end{bmatrix}, \quad (19.32)$$

and

$$f^2(n) = \begin{cases} 10(n)^2, & n \geq 0 \\ 0\,, & n < 0 \end{cases}. \quad (19.33)$$

The equations of operation of the layer will be

$$(0.1)\frac{dn_1^2(t)}{dt} = -n_1^2(t) + (1 - n_1^2(t))\{f^2(n_1^2(t)) + ({}_1\mathbf{w}^{1:2})^T\mathbf{a}^1\} - (n_1^2(t) + 1)f^2(n_2^2(t)) \quad (19.34)$$

$$(0.1)\frac{dn_2^2(t)}{dt} = -n_2^2(t) + (1 - n_2^2(t))\{f^2(n_2^2(t)) + ({}_2\mathbf{w}^{1:2})^T\mathbf{a}^1\} - (n_2^2(t) + 1)f^2(n_1^2(t))\,. \quad (19.35)$$

This is identical in form to the Grossberg Layer 2 example in Chapter 18 (see Eq. (18.20) and Eq. (18.21)), except that ${}^{-}b^2 = 1$. This will allow $n_1^2(t)$ and $n_2^2(t)$ to range between -1 and +1.

The inputs to Layer 2 are the inner products of the prototype patterns (rows of the weight matrix $\mathbf{W}^{1:2}$) with the output of Layer 1. (The rows of this weight matrix are normalized, as will be explained in a later section.) The largest inner product will correspond to the prototype pattern that is closest to the output of Layer 1. Layer 2 then performs a competition between the neurons. The transfer function $f^2(n)$ is chosen to be a faster-than-linear transfer function (see Chapter 18, page 18-20, for a discussion of the effect of $f^2(n)$). This choice will force the neuron with largest input to have a positive n, and the other neuron to have a negative n (with appropriate choice of network parameters). After the competition, one neuron output will be 1, and the other neuron output will be zero, since we are using the $hardlim^+$ transfer function to compute the layer output.

Figure 19.5 illustrates the response of Layer 2 when the input vector is $\mathbf{a}^1 = \begin{bmatrix}1 & 0\end{bmatrix}^T$. The second row of $\mathbf{W}^{1:2}$ has a larger inner product with $\mathbf{a}^1$

than the second row, therefore neuron 2 wins the competition. At steady state, $n_2^2(t)$ has a positive value, and $n_1^2(t)$ has a negative value. The steady state Layer 2 output will then be

$$\mathbf{a}^2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{19.36}$$

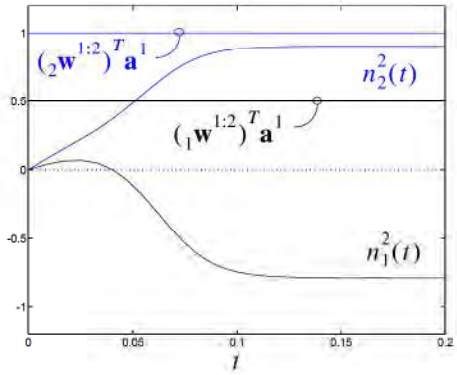


Figure 19.5 Response of Layer 2

We can summarize the steady state operation of Layer 2 as follows:

$$a_i^2 = \begin{cases} 1, & \text{if}(({}_i\mathbf{w}^{1:2})^T\mathbf{a}^1 = max[({}_j\mathbf{w}^{1:2})^T\mathbf{a}^1]) \\ 0, & \text{otherwise} \end{cases}. \tag{19.37}$$



To experiment with Layer 2 of the ART1 network, use the Neural Network Design Demonstration ART1 Layer 2 (`nnd16al2`).

Orienting Subsystem

One of the key elements of the ART architecture is the Orienting Subsystem. Its purpose is to determine if there is a sufficient match between the L2-L1 expectation and the input pattern. When there is not enough of a match, the Orienting Subsystem should send a reset signal to Layer 2. The reset signal will cause a long-lasting inhibition of the previous winning neuron, and thus allow another neuron to win the competition.

Figure 19.6 displays the Orienting Subsystem.

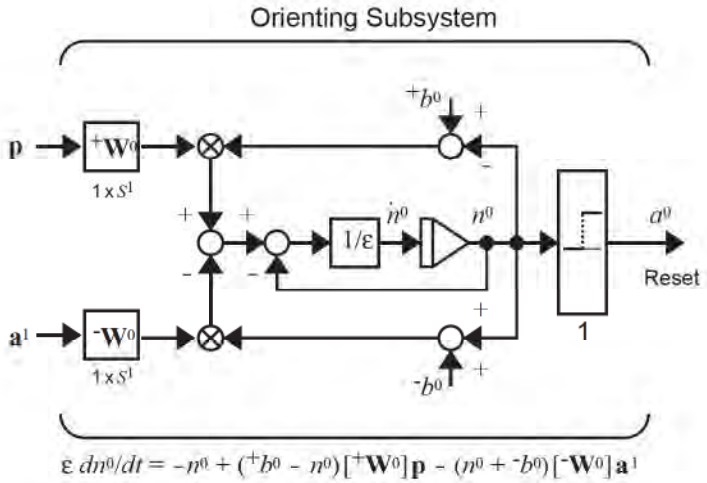


Figure 19.6 Orienting Subsystem of the ART1 Network

The equation of operation of the Orienting Subsystem is

$$\varepsilon \frac{dn^0(t)}{dt} = -n^0(t) + ({}^+b^0 - n^0(t))\{{}^+\mathbf{W}^0\mathbf{p}\} - (n^0(t) + {}^-b^0)\{{}^-\mathbf{W}^0\mathbf{a}^1\}. \quad (19.38)$$

This is a shunting model, with excitatory input ${}^+\mathbf{W}^0\mathbf{p}$, where

$${}^+\mathbf{W}^0 = [\alpha\ \alpha\ \ldots\ \alpha]. \quad (19.39)$$

Therefore, the excitatory input can be written

$${}^+\mathbf{W}^0\mathbf{p} = [\alpha\ \alpha\ \ldots\ \alpha]\mathbf{p} = \alpha \sum_{j=1}^{S^1} p_j = \alpha \|\mathbf{p}\|^2, \quad (19.40)$$

where the last equality holds because $\mathbf{p}$ is a binary vector.

The inhibitory input to the Orienting Subsystem is ${}^-\mathbf{W}^0\mathbf{a}^1$, where

$${}^-\mathbf{W}^0 = [\beta\ \beta\ \ldots\ \beta]. \quad (19.41)$$

Therefore, the inhibitory input can be written

$${}^-\mathbf{W}^0\mathbf{a}^1 = [\beta\ \beta\ \ldots\ \beta]\mathbf{a}^1 = \beta \sum_{j=1}^{S^1} a_j^1(t) = \beta \|\mathbf{a}^1\|^2. \quad (19.42)$$

Whenever the excitatory input is larger than the inhibitory input, the Orienting Subsystem will be driven on. Consider the following steady state operation:

$$\begin{aligned} 0 &= -n^0 + ({}^+b^0 - n^0)\{\alpha\|\mathbf{p}\|^2\} - (n^0 + {}^-b^0)\{\beta\|\mathbf{a}^1\|^2\} \\ &= -(1 + \alpha\|\mathbf{p}\|^2 + \beta\|\mathbf{a}^1\|^2)n^0 + {}^+b^0(\alpha\|\mathbf{p}\|^2) - {}^-b^0(\beta\|\mathbf{a}^1\|^2) . \end{aligned} \tag{19.43}$$

If we solve for n^0, we find

$$n^0 = \frac{{}^+b^0(\alpha\|\mathbf{p}\|^2) - {}^-b^0(\beta\|\mathbf{a}^1\|^2)}{(1 + \alpha\|\mathbf{p}\|^2 + \beta\|\mathbf{a}^1\|^2)} . \tag{19.44}$$

Let ${}^+b^0 = {}^-b^0 = 1$, then $n^0 > 0$ if $\alpha\|\mathbf{p}\|^2 - \beta\|\mathbf{a}^1\|^2 > 0$, or in other words:

$$n^0 > 0 \text{ if } \frac{\|\mathbf{a}^1\|^2}{\|\mathbf{p}\|^2} < \frac{\alpha}{\beta} = \rho . \tag{19.45}$$

Vigilance

This is the condition that will cause a reset of Layer 2, since $a^0 = hardlim^+(n^0)$. The term ρ is called the *vigilance* parameter, and must fall in the range $0 < \rho < 1$. If the vigilance is close to 1, a reset will occur unless $\mathbf{a}^1$ is close to $\mathbf{p}$. If the vigilance is close to 0, $\mathbf{a}^1$ need not be close to $\mathbf{p}$ to prevent a reset. The vigilance parameter determines the coarseness of the categorization (or clustering) created by the prototype vectors.

Recall from Eq. (19.21) that $\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1}$ whenever Layer 2 is active. Therefore, $\|\mathbf{p}\|^2$ will always be greater than or equal to $\|\mathbf{a}^1\|^2$. They will be equal when the expectation $\mathbf{w}_j^{2:1}$ has a 1 wherever the input $\mathbf{p}$ has a 1. Therefore, the orienting subsystem will cause a reset when there is enough of a mismatch between $\mathbf{p}$ and $\mathbf{w}_j^{2:1}$. The amount of mismatch required for a reset is determined by the vigilance parameter ρ.



To demonstrate the operation of the Orienting Subsystem, suppose that $\varepsilon = 0.1$, $\alpha = 3$, $\beta = 4$ ($\rho = 0.75$),

$$\mathbf{p} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{a}^1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} . \tag{19.46}$$

The equation of operation becomes

$$\begin{aligned} (0.1)\frac{dn^0(t)}{dt} &= -n^0(t) + (1 - n^0(t))\{3(p_1 + p_2)\} \\ &\quad - (n^0(t) + 1)\{4(a_1^1 + a_2^1)\} \end{aligned} \tag{19.47}$$

or

$$\frac{dn^0(t)}{dt} = -110n^0(t) + 20. \tag{19.48}$$

The response is plotted in Figure 19.7. In this case a reset signal will be sent to Layer 2, since $n^0(t)$ is positive. In this example, because the vigilance parameter is set to $\rho = 0.75$, and $\mathbf{p}$ has only two elements, we will have a reset whenever $\mathbf{p}$ and $\mathbf{a}^1$ are not identical. (If the vigilance parameter were set to $\rho = 0.25$, we would not have had a reset for the $\mathbf{p}$ and $\mathbf{a}^1$ of Eq. (19.46), since $\|\mathbf{a}^1\|^2/\|\mathbf{p}\|^2 = 1/2$.)

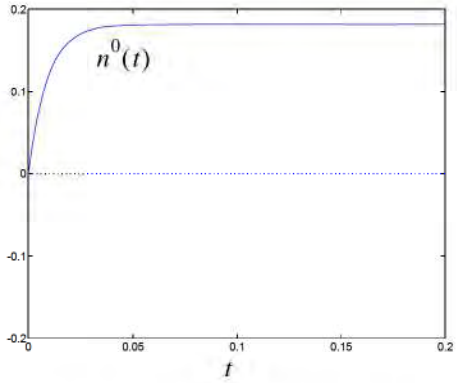


Figure 19.7 Response of the Orienting Subsystem

The steady state operation of the Orienting Subsystem can be summarized as follows:

$$a^0 = \begin{cases} 1, & \text{if}[\|\mathbf{a}^1\|^2/\|\mathbf{p}\|^2 < \rho] \\ 0, & \text{otherwise} \end{cases}. \tag{19.49}$$



To experiment with the Orienting Subsystem, use the Neural Network Design Demonstration Orienting Subsystem (**nnd16os**).

Learning Law: L1-L2

The ART1 network has two separate learning laws: one for the L1-L2 connections, and another for the L2-L1 connections. The L1-L2 connections use a type of instar learning to learn to *recognize* a set of prototype patterns. The L2-L1 connections use outstar learning in order to *reproduce* (or recall) a set of prototype patterns. In this section we will describe the L1-

L2 instar learning law, and in the following section we will present the L2-L1 outstar learning law.

We should note that the L1-L2 connections and the L2-L1 connections are updated at the same time. Whenever the input pattern and the expectation have an adequate match, as determined by the Orienting Subsystem, both $\mathbf{W}^{1:2}$ and $\mathbf{W}^{2:1}$ are adapted. This process of matching, and subsequent adaptation, is referred to as *resonance*, hence the name adaptive resonance theory.

Resonance

Subset/Superset Dilemma

The learning in the L1-L2 connections of the ART1 network is very close to the learning in the Grossberg network of Chapter 18, with one major difference. In the Grossberg network, the input patterns are normalized in Layer 1, and therefore all of the prototype patterns will have the same length. In the ART1 network no normalization takes place in Layer 1. Therefore a problem can occur when one prototype pattern is a subset of another. For example, suppose that the L1-L2 connection matrix is

$$\mathbf{W}^{1:2} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \tag{19.50}$$

so that the prototype patterns are

$${}_1\mathbf{w}^{1:2} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \text{ and } {}_2\mathbf{w}^{1:2} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \tag{19.51}$$

We say that ${}_1\mathbf{w}^{1:2}$ is a subset of ${}_2\mathbf{w}^{1:2}$, since ${}_2\mathbf{w}^{1:2}$ has a 1 wherever ${}_1\mathbf{w}^{1:2}$ has a 1.

If the output of Layer 1 is

$$\mathbf{a}^1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \tag{19.52}$$

then the input to Layer 2 will be

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}. \tag{19.53}$$

Both prototype vectors have the same inner product with $\mathbf{a}^1$, even though the first prototype is identical to $\mathbf{a}^1$ and the second prototype is not. This is called the subset/superset dilemma.

One solution to the subset/superset dilemma is to normalize the prototype patterns. That is, when a prototype pattern has a large number of nonzero entries, the magnitude of each entry should be reduced. For example, using our preceding problem, we could modify the L1-L2 matrix as follows:

$$\mathbf{W}^{1:2} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}. \tag{19.54}$$

The input to Layer 2 will then be

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{2}{3} \end{bmatrix}. \tag{19.55}$$

Now we have the desired result: the first prototype has the largest inner product with $\mathbf{a}^1$. The first neuron in Layer 2 will be activated.

In the Grossberg network of Chapter 18 we obtained normalized prototype patterns by normalizing the input patterns in Layer 1. In the ART1 network we will normalize the prototype patterns by using an on-center/off-surround competition in the L1-L2 learning law.

Learning Law

The learning law for $\mathbf{W}^{1:2}$ is

$$\frac{d[{}_i\mathbf{w}^{1:2}(t)]}{dt} = a_i^2(t)[\{{}^+\mathbf{b} - {}_i\mathbf{w}^{1:2}(t)\}\zeta[{}^+\mathbf{W}]\mathbf{a}^1(t) - \{{}_i\mathbf{w}^{1:2}(t) + {}^-\mathbf{b}\}[{}^-\mathbf{W}]\mathbf{a}^1(t)], \tag{19.56}$$

where

$${}^+\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \quad {}^-\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad {}^+\mathbf{W} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \text{ and } {}^-\mathbf{W} = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix}. \tag{19.57}$$

This is a modified form of instar learning. When neuron i of Layer 2 is active, the ith row of $\mathbf{W}^{1:2}$, ${}_i\mathbf{w}^{1:2}$, is moved in the direction of $\mathbf{a}^1$. The difference between Eq. (19.56) and the standard instar learning is that the elements of ${}_i\mathbf{w}^{1:2}$ compete, and therefore ${}_i\mathbf{w}^{1:2}$ is normalized. In the bracket on the right side of Eq. (19.56) we can see that it has the form of a shunting model, with on-center/off-surround input connections from $\mathbf{a}^1$. The excitatory bias is ${}^+\mathbf{b} = \mathbf{1}$ (a vector of 1's), and the inhibitory bias is ${}^-\mathbf{b} = \mathbf{0}$, which ensures that the elements of ${}_i\mathbf{w}^{1:2}$ remain between 0 and 1. (Recall our discussion of the shunting model in Chapter 18.)

To verify that Eq. (19.56) causes normalization of the prototype patterns, let's investigate the steady state operation. For this analysis we will assume that the outputs of Layer 1 and Layer 2 remain constant until the weights reach steady state. This is called *fast learning*.

Fast Learning

The equation for element $w_{i,j}^{1:2}$ is

$$\frac{dw_{i,j}^{1:2}(t)}{dt} = a_i^2(t)\left[(1 - w_{i,j}^{1:2}(t))\zeta a_j^1(t) - w_{i,j}^{1:2}(t)\sum_{k \neq j} a_k^1(t)\right]. \tag{19.58}$$

If we assume that neuron i is active in Layer 2 ($a_i^2(t) = 1$) and set the derivative to zero in Eq. (19.58), we see that

$$0 = \left[(1 - w_{i,j}^{1:2})\zeta a_j^1 - w_{i,j}^{1:2}\sum_{k \neq j} a_k^1\right]. \tag{19.59}$$

To find the steady state value of $w_{i,j}^{1:2}$, we will consider two cases. First, assume that $a_j^1 = 1$. Then we have

$$0 = (1 - w_{i,j}^{1:2})\zeta - w_{i,j}^{1:2}(\|\mathbf{a}^1\|^2 - 1) = -(\zeta + \|\mathbf{a}^1\|^2 - 1)w_{i,j}^{1:2} + \zeta, \tag{19.60}$$

or

$$w_{i,j}^{1:2} = \frac{\zeta}{\zeta + \|\mathbf{a}^1\|^2 - 1}. \tag{19.61}$$

(Note that $\sum_{k=1}^{S^1} a_k^1 = \|\mathbf{a}^1\|^2$, since $\mathbf{a}^1$ is a binary vector.)

On the other hand, if $a_j^1 = 0$, then Eq. (19.59) reduces to

$$0 = -w_{i,j}^{1:2}\|\mathbf{a}^1\|^2, \tag{19.62}$$

or

$$w_{i,j}^{1:2} = 0 . \tag{19.63}$$

To summarize Eq. (19.61) and Eq. (19.63):

$$_i\mathbf{w}^{1:2} = \frac{\zeta \mathbf{a}^1}{\zeta + \left\|\mathbf{a}^1\right\|^2 - 1}, \tag{19.64}$$

where $\zeta > 1$ to ensure that the denominator will never equal zero.

Therefore the prototype patterns will be normalized, and this will solve the subset/superset dilemma. (By "normalized" here we do not mean that all prototype vectors will have unit length in Euclidean distance, but simply that the rows of $\mathbf{W}^{1:2}$ that have more nonzero entries will have elements with smaller magnitudes. In this case, vectors with more nonzero entries may actually have a smaller length than vectors with fewer nonzero entries.)

Learning Law: L2-L1

The L2-L1 connections, $\mathbf{W}^{2:1}$, in the ART1 architecture are trained using an outstar learning rule. The purpose of the L2-L1 connections is to recall an appropriate prototype pattern (the expectation), so that it can be compared and combined, in Layer 1, with the input pattern. When the expectation and the input pattern do not match, a reset is sent to Layer 2, so that a new prototype pattern can be chosen (as we have discussed in previous sections).

The learning law for $\mathbf{W}^{2:1}$ is a typical outstar equation:

$$\frac{d[\mathbf{w}_j^{2:1}(t)]}{dt} = a_j^2(t)[-\mathbf{w}_j^{2:1}(t) + \mathbf{a}^1(t)] . \tag{19.65}$$

Therefore, if neuron j in Layer 2 is active (has won the competition), then column j of $\mathbf{W}^{2:1}$ is moved toward the $\mathbf{a}^1$ pattern. To illustrate this, let's investigate the steady state operation of Eq. (19.65).

For this analysis we will assume the fast learning scenario, where the outputs of Layer 1 and Layer 2 remain constant until the weights reach steady state. Assume that neuron j in Layer 2 is active, so that $a_j^2 = 1$. Setting the derivative in Eq. (19.65) to zero, we find

$$\mathbf{0} = -\mathbf{w}_j^{2:1} + \mathbf{a}^1 , \text{ or } \mathbf{w}_j^{2:1} = \mathbf{a}^1 . \tag{19.66}$$

Therefore column j of $\mathbf{W}^{2:1}$ converges to the output of Layer 1, $\mathbf{a}^1$. Recall from Eq. (19.20) and Eq. (19.21) that $\mathbf{a}^1$ is a combination of the input pattern and the appropriate prototype pattern. Therefore the prototype pat-

tern is modified to incorporate the current input pattern (if there is a close enough match).

Keep in mind that $\mathbf{W}^{1:2}$ and $\mathbf{W}^{2:1}$ are updated at the same time. When neuron j of Layer 2 is active and there is a sufficient match between the expectation and the input pattern (which indicates a resonance condition), then row j of $\mathbf{W}^{1:2}$ and column j of $\mathbf{W}^{2:1}$ are adapted. In fast learning, column j of $\mathbf{W}^{2:1}$ is set to $\mathbf{a}^1$, while row j of $\mathbf{W}^{1:2}$ is set to a normalized version of $\mathbf{a}^1$.

ART1 Algorithm Summary

Now that we have investigated each of the subsystems of the ART1 architecture, we can gain some insight into its overall operation if we summarize the key steady state equations and organize them into an algorithm.

Initialization

The ART1 algorithm begins with an initialization of the weight matrices $\mathbf{W}^{1:2}$ and $\mathbf{W}^{2:1}$. The initial $\mathbf{W}^{2:1}$ matrix is set to all 1's. Thus, the first time a new neuron in Layer 2 wins a competition, resonance will occur, since $\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1} = \mathbf{p}$ and therefore $\|\mathbf{a}^1\|^2 / \|\mathbf{p}\|^2 = 1 > \rho$. This means that any untrained column in $\mathbf{W}^{2:1}$ is effectively a blank slate and will cause a match with any input pattern.

Since the rows of the $\mathbf{W}^{1:2}$ matrix should be normalized versions of the columns of $\mathbf{W}^{2:1}$, every element of the initial $\mathbf{W}^{1:2}$ matrix is set to $\zeta/(\zeta + S^1 - 1)$.

Algorithm

After initialization, the ART1 algorithm proceeds as follows:

1. First, we present an input pattern to the network. Since Layer 2 is not active on initialization (i.e., each $a_j^2 = 0$), the output of Layer 1 is (Eq. (19.20))

$$\mathbf{a}^1 = \mathbf{p}. \tag{19.67}$$

2. Next, we compute the input to Layer 2,

$$\mathbf{W}^{1:2}\mathbf{a}^1, \tag{19.68}$$

and activate the neuron in Layer 2 with the largest input (Eq. (19.37)):

$$a_i^2 = \begin{cases} 1, & \text{if}((_i\mathbf{w}^{1:2})^T\mathbf{a}^1 = max[(_k\mathbf{w}^{1:2})^T\mathbf{a}^1]) \\ 0, & \text{otherwise} \end{cases}. \tag{19.69}$$

In case of a tie, the neuron with the smallest index is declared the winner.

3. We then compute the L2-L1 expectation (where we assume neuron j of Layer 2 is activated):

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \mathbf{w}_j^{2:1}. \tag{19.70}$$

4. Now that Layer 2 is active, we adjust the Layer 1 output to include the L2-L1 expectation (Eq. (19.21)):

$$\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1}. \tag{19.71}$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern (Eq. (19.49)):

$$a^0 = \begin{cases} 1, & \text{if}[\|\mathbf{a}^1\|^2 / \|\mathbf{p}\|^2 < \rho] \\ 0, & \text{otherwise} \end{cases}. \tag{19.72}$$

6. If $a^0 = 1$, then we set $a_j^2 = 0$, inhibit it until an adequate match occurs (resonance), and return to step 1. If $a^0 = 0$, we continue with step 7.

7. Resonance has occurred. Therefore we update row j of $\mathbf{W}^{1:2}$ (Eq. (19.61)):

$$_j\mathbf{w}^{1:2} = \frac{\zeta\mathbf{a}^1}{\zeta + \|\mathbf{a}^1\|^2 - 1}. \tag{19.73}$$

8. We now update column j of $\mathbf{W}^{2:1}$ (Eq. (19.66)):

$$\mathbf{w}_j^{2:1} = \mathbf{a}^1. \tag{19.74}$$

9. We remove the input pattern, restore all inhibited neurons in Layer 2, and return to step 1 with a new input pattern.

The input patterns continue to be applied to the network until the weights stabilize (do not change). Carpenter and Grossberg have shown [CaGr87a] that the ART1 algorithm will always form stable clusters for any set of input patterns.

See Problems P19.5, P19.6 and P19.7 for detailed examples of the ART1 algorithm.



To experiment with the ART1 algorithm, use the Neural Network Design Demonstration ART1 (`nnd16a1`).

Other ART Architectures

The ART1 network is just one example of adaptive resonance theory. Carpenter and Grossberg, and others in their research group, have developed many variations on this theme.

One disadvantage of the ART1 network is that it can only be used for binary input patterns. Carpenter and Grossberg developed a variation of ART1, called ART2, to handle either analog or binary patterns [CaGr87b]. The basic structure of ART2 is very similar to ART1, with the exception of Layer 1. In ART2 several sublayers take the place of Layer 1. These sublayers are needed because analog vectors, unlike binary vectors, can be arbitrarily close together. The sublayers perform a combination of normalization and noise suppression, in addition to the comparison of the input vector and the expectation that is needed by the orienting subsystem.

Carpenter and Grossberg later developed the ART3 network [CaGr90], which introduced a more sophisticated biological model for the reset mechanism required for ART. Up to the present time, this network has not been widely applied.

In 1991 Carpenter, Grossberg and Reynolds introduced the ARTMAP network [CaGrRe91]. In contrast with all of the previous ART networks, it is a supervised network. The ARTMAP architecture consists of two ART modules that are connected by an "inter-ART" associative memory. One ART module receives the input vector, while the other ART module receives the desired output vector. The network learns to predict the correct output vector whenever the input vector is presented.

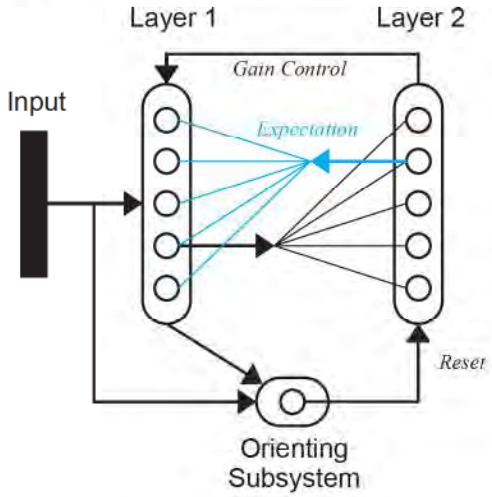
More recently, Carpenter, Grossberg, Markuzon, Reynolds and Rosen have modified the ARTMAP architecture to incorporate fuzzy logic. The result is referred to as Fuzzy ARTMAP [CaGrMa92]. It seems to improve performance, especially with noisy input patterns.

All of these ART architectures incorporate the key modules discussed in this chapter, including:

- L1-L2 instars for pattern recognition.
- L2-L1 outstars for pattern recall.
- Layer 2 for contrast enhancement (competition).
- Layer 1 for comparison of input and expectation.
- Orienting Subsystem for resetting when a pattern mismatch occurs.

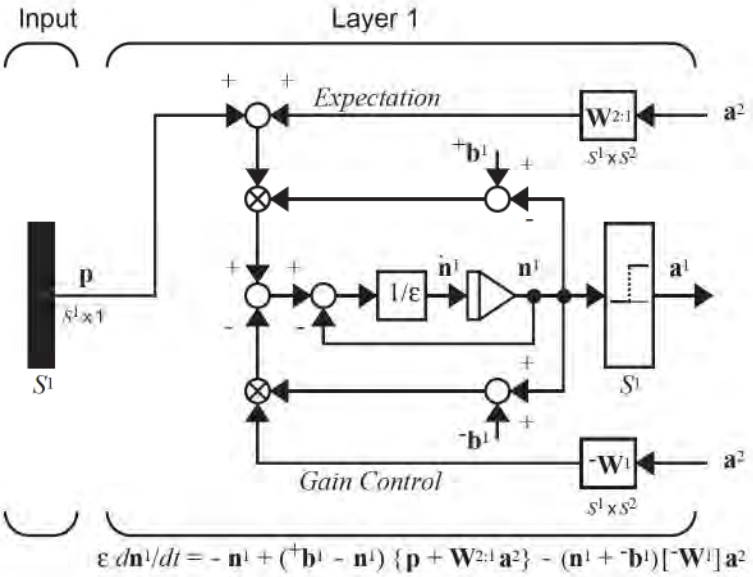
Summary of Results

Basic ART Architecture



ART1 Network (Binary Patterns)

ART1 Layer 1



Layer 1 Equation

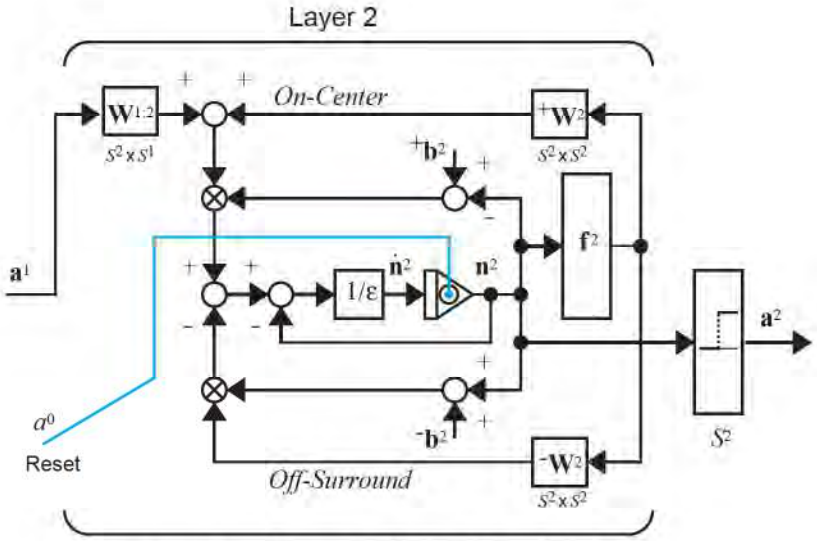
$$\varepsilon \frac{d\mathbf{n}^1(t)}{dt} = -\mathbf{n}^1(t) + ({}^{+}\mathbf{b}^1 - \mathbf{n}^1(t))\{\mathbf{p} + \mathbf{W}^{2:1}\mathbf{a}^2(t)\} - (\mathbf{n}^1(t) + {}^{-}\mathbf{b}^1)[{}^{-}\mathbf{W}^1]\mathbf{a}^2(t)$$

Steady State Operation

If Layer 2 is not active (i.e., each $a_j^2 = 0$), $\mathbf{a}^1 = \mathbf{p}$.

If Layer 2 is active (i.e., one $a_j^2 = 1$), $\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1}$.

ART1 Layer 2



$$\varepsilon\, d\mathbf{n}^2/dt = -\mathbf{n}^2 + ({}^{+}\mathbf{b}^2 - \mathbf{n}^2)\{[{}^{+}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2) + \mathbf{W}^{1:2}\mathbf{a}^1\} - (\mathbf{n}^2 + {}^{-}\mathbf{b}^2)[{}^{-}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2)$$

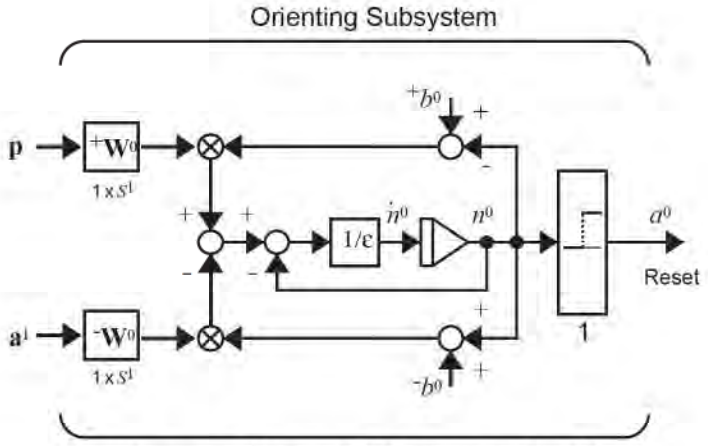
Layer 2 Equation

$$\varepsilon \frac{d\mathbf{n}^2(t)}{dt} = -\mathbf{n}^2(t) + ({}^{+}\mathbf{b}^2 - \mathbf{n}^2(t))\{[{}^{+}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t)) + \mathbf{W}^{1:2}\mathbf{a}^1\} - (\mathbf{n}^2(t) + {}^{-}\mathbf{b}^2)[{}^{-}\mathbf{W}^2]\mathbf{f}^2(\mathbf{n}^2(t))$$

Steady State Operation

$$a_i^2 = \begin{cases} 1, & \text{if}\,(({}_i\mathbf{w}^{1:2})^T\mathbf{a}^1 = max[({}_j\mathbf{w}^{1:2})^T\mathbf{a}^1]) \\ 0, & \text{otherwise} \end{cases}$$

Orienting Subsystem



Orienting Subsystem Equation

$$\varepsilon \frac{dn^{0}(t)}{dt} = -n^{0}(t) + (^{+}b^{0} - n^{0}(t))\{^{+}\mathbf{W}^{0}\mathbf{p}\} - (n^{0}(t) + {}^{-}b^{0})\{^{-}\mathbf{W}^{0}\mathbf{a}^{1}\}$$

where $^{+}\mathbf{W}^{0} = [\alpha\ \alpha\ \ldots\ \alpha]$, $^{-}\mathbf{W}^{0} = [\beta\ \beta\ \ldots\ \beta]$, $^{+}b^{0} = {}^{-}b^{0} = 1$

Steady State Operation

$$a^{0} = \begin{cases} 1, & \text{if}[\|\mathbf{a}^{1}\|^{2}/\|\mathbf{p}\|^{2} < \rho] \\ 0, & \text{otherwise} \end{cases}$$

L1-L2 Learning Law

$$\frac{d[{}_{i}\mathbf{w}^{1:2}(t)]}{dt} = a_{i}^{2}(t)[\{^{+}\mathbf{b} - {}_{i}\mathbf{w}^{1:2}(t)\}\zeta[^{+}\mathbf{W}]\mathbf{a}^{1}(t) - \{{}_{i}\mathbf{w}^{1:2}(t) + {}^{-}\mathbf{b}\}[^{-}\mathbf{W}]\mathbf{a}^{1}(t)]$$

$$^{+}\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad ^{-}\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad ^{+}\mathbf{W} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \quad ^{-}\mathbf{W} = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix}$$

Steady State Operation (Fast Learning)

$$ {}_i\mathbf{w}^{1:2} = \frac{\zeta \mathbf{a}^1}{\zeta + \|\mathbf{a}^1\|^2 - 1} \quad \text{(Neuron } i \text{ in Layer 2 Active)} $$

L2-L1 Learning Law

$$ \frac{d[\mathbf{w}_j^{2:1}(t)]}{dt} = a_j^2(t)[-\mathbf{w}_j^{2:1}(t) + \mathbf{a}^1(t)] $$

Steady State Operation (Fast Learning)

$$ \mathbf{w}_j^{2:1} = \mathbf{a}^1 \quad \text{(Neuron } j \text{ in Layer 2 Active)} $$

ART1 Algorithm (Fast Learning) Summary

Initialization

The initial $\mathbf{W}^{2:1}$ matrix is set to all 1's.

Every element of the initial $\mathbf{W}^{1:2}$ matrix is set to $\zeta/(\zeta + S^1 - 1)$.

Algorithm

1. First, we present an input pattern to the network. Since Layer 2 is not active on initialization (i.e., each $a_j^2 = 0$), the output of Layer 1 is

$$ \mathbf{a}^1 = \mathbf{p}. $$

2. Next, we compute the input to Layer 2,

$$ \mathbf{W}^{1:2}\mathbf{a}^1, $$

and activate the neuron in Layer 2 with the largest input:

$$ a_i^2 = \begin{cases} 1, & \text{if}({}_i\mathbf{w}^{1:2})^T\mathbf{a}^1 = max[({}_k\mathbf{w}^{1:2})^T\mathbf{a}^1]) \\ 0, & \text{otherwise} \end{cases}. $$

In case of a tie, the neuron with the smallest index is declared the winner.

3. We then compute the L2-L1 expectation (where we assume neuron j of Layer 2 is activated):

$$ \mathbf{W}^{2:1}\mathbf{a}^2 = \mathbf{w}_j^{2:1}. $$

4. Now that Layer 2 is active, we adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p} \cap \mathbf{w}_j^{2:1} .$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$a^0 = \begin{cases} 1 . & \text{if}[\|\mathbf{a}^1\|^2 / \|\mathbf{p}\|^2 < \rho] \\ 0 , & \text{otherwise} \end{cases} .$$

6. If $a^0 = 1$, then we set $a_j^2 = 0$, inhibit it until an adequate match occurs (resonance), and return to step 1. If $a^0 = 0$, we continue with step 7.

7. Resonance has occurred, therefore we update row j of $\mathbf{W}^{1:2}$:

$${}_j\mathbf{w}^{1:2} = \frac{\zeta \mathbf{a}^1}{\zeta + \|\mathbf{a}^1\|^2 - 1} .$$

8. We now update column j of $\mathbf{W}^{2:1}$:

$$\mathbf{w}_j^{2:1} = \mathbf{a}^1 .$$

9. We remove the input pattern, restore all inhibited neurons in Layer 2, and return to step 1 with a new input pattern.

Solved Problems

P19.1 Consider Layer 1 of the ART1 network with the following parameters:

$$\varepsilon = 0.01 \quad {}^{+}b^{1} = 2 \quad {}^{-}b^{1} = 3 .$$

Assume two neurons in Layer 2, two elements in the input vector and the following weight matrix and input:

$$\mathbf{W}^{2:1} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} .$$

Also, assume that neuron 1 of Layer 2 is active.

i. Find and plot the response $\mathbf{n}^1$.

ii. Check to see that the answer to part (i) satisfies the steady state response predicted by Eq. (19.21).

i. Since Layer 2 is active, and neuron 1 of Layer 2 wins the competition, the equations of operation of Layer 1 are

$$(0.01)\frac{dn_1^1}{dt} = -n_1^1 + (2 - n_1^1)\{p_1 + w_{1,1}^{2:1}\} - (n_1^1 + 3)$$

$$= -n_1^1 + (2 - n_1^1)\{1 + 0\} - (n_1^1 + 3) = -3n_1^1 - 1$$

$$(0.01)\frac{dn_2^1}{dt} = -n_2^1 + (2 - n_2^1)\{p_2 + w_{2,1}^{2:1}\} - (n_2^1 + 3)$$

$$= -n_2^1 + (2 - n_2^1)\{1 + 1\} - (n_2^1 + 3) = -4n_2^1 + 1 .$$

These can be simplified to obtain

$$\frac{dn_1^1}{dt} = -300n_1^1 - 100 ,$$

$$\frac{dn_2^1}{dt} = -400n_2^1 + 100 .$$

If we assume that both neurons start with zero initial condition, the solutions are

$$n_1^1(t) = -\frac{1}{3}[1 - e^{-300t}],$$

$$n_2^1(t) = \frac{1}{4}[1 - e^{-400t}].$$

These are displayed in Figure P19.1.

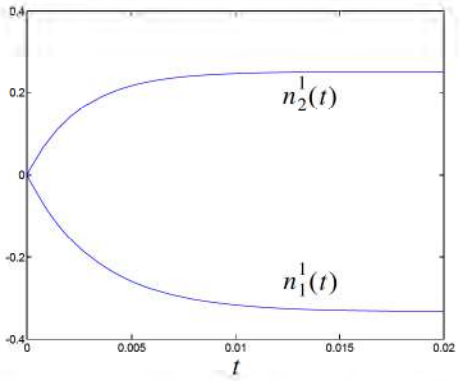


Figure P19.1 Response of Layer 1

ii. Note that $n_1^1(t)$ converges to a negative value, and $n_2^1(t)$ converges to a positive value. Therefore, $a_1^1(t)$ converges to 0, and $a_2^1(t)$ converges to 1 (recall that the transfer function for Layer 1 is $hardlim^+$). This agrees with our steady state analysis (see Eq. (19.21)), since

$$\mathbf{p} \cap \mathbf{w}_1^{2:1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cap \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \mathbf{a}^1 . \tag{19.75}$$

P19.2 Consider Layer 2 of the ART1 network with the following parameters:

$$\varepsilon = 0.1 \quad {}^{+}\mathbf{b}^2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad {}^{-}\mathbf{b}^2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad \mathbf{W}^{1:2} = \begin{bmatrix} ({}_1\mathbf{w}^{1:2})^T \\ ({}_2\mathbf{w}^{1:2})^T \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 1 & 0 \end{bmatrix}$$

and

$$f^2(n) = \begin{cases} 10(n)^2 & n \geq 0 \\ 0 & n < 0 \end{cases}.$$

Assume that the output of Layer 1 is

$$\mathbf{a}^1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

This is equivalent to the Layer 2 example in the text (page 19-12), with the exception of the bias values.

i. Write the equations of operation of Layer 2 and simulate and plot the response. Explain the effect of increasing the bias values.

ii. Verify that the steady state operation of Layer 2 is correct.

i. The equations of operation of the layer will be

$$(0.1)\frac{dn_1^2(t)}{dt} = -n_1^2(t) + (2 - n_1^2(t))\{f^2(n_1^2(t)) + ({}_1\mathbf{w}^{1:2})^T\mathbf{a}^1\} - (n_1^2(t) + 2)f^2(n_2^2(t)),$$

$$(0.1)\frac{dn_2^2(t)}{dt} = -n_2^2(t) + (2 - n_2^2(t))\{f^2(n_2^2(t)) + ({}_2\mathbf{w}^{1:2})^T\mathbf{a}^1\} - (n_2^2(t) + 2)f^2(n_1^2(t)).$$

Figure P19.2 illustrates the response of Layer 2 when the input vector is $\mathbf{a}^1 = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$. The second row of $\mathbf{W}^{1:2}$ has a larger inner product with $\mathbf{a}^1$ than the first row, therefore neuron 2 wins the competition.

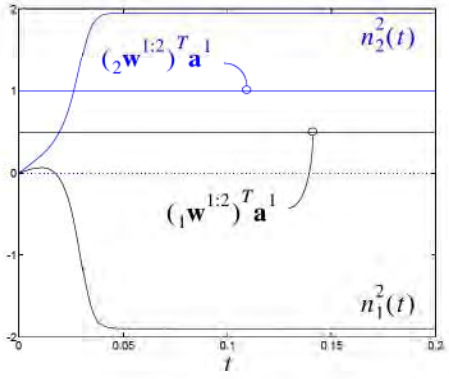


Figure P19.2 Response of Layer 2

If we compare Figure P19.2 with Figure 19.5, we can see that the bias value has three effects. First, the speed of response is increased; the neuron outputs move more quickly to their steady state values. Second, the range of the response is increased from [–1, 1] to [–2, 2]. (Recall from Chapter 18 that for the shunting model the upper limit will be the excitatory bias ${}^{+}b$. The lower limit will be the inhibitory bias ${}^{-}b$.) Third, the neuron responses move closer to the upper and lower limits.

ii. At steady state, $n_1^2(t)$ has a positive value, and $n_2^2(t)$ has a negative value. The steady state Layer 2 output will then be

$$\mathbf{a}^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

This agrees with the desired steady state response characteristics of Layer 2:

$$a_i^2 = \begin{cases} 1, & \text{if}(({}_i\mathbf{w}^{1:2})^T\mathbf{a}^1 = max[({}_j\mathbf{w}^{1:2})^T\mathbf{a}^1]) \\ 0, & \text{otherwise} \end{cases}.$$

P19.3 Consider the Orienting Subsystem of the ART1 network with the following parameters:

$$\varepsilon = 0.1 \quad \alpha = 0.5 \quad \beta = 2 \quad (\rho = 0.25) \quad {}^{+}b^0 = {}^{-}b^0 = 0.5.$$

The inputs to the Orienting Subsystem are

$$\mathbf{p} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{a}^1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

i. Find and plot the response of the Orienting Subsystem $n^0(t)$.

ii. Verify that the steady state conditions are satisfied.

The equation of operation of the Orienting Subsystem is

$$(0.1)\frac{dn^0(t)}{dt} = -n^0(t) + (0.5 - n^0(t))\{0.5(p_1 + p_2 + p_3)\} - (n^0(t) + 0.5)\{2(a_1^1 + a_2^1 + a_3^1)\}$$

or

$$\frac{dn^0(t)}{dt} = -65n^0(t) - 12.5 .$$

The response is then

$$n^0(t) = -0.1923[1 - e^{-65t}]$$

This response is plotted in Figure P19.3. In this case, since $n^0(t)$ is negative, $a^0 = hardlim^+(n^0) = 0$, and therefore a reset signal will not be sent to Layer 2.

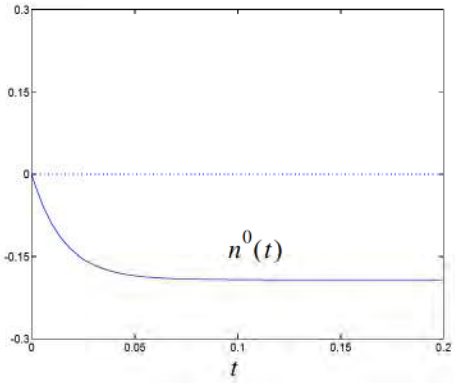


Figure P19.3 Response of the Orienting Subsystem

ii. The steady state operation of the Orienting Subsystem can be summarized as follows:

$$a^0 = \begin{cases} 1 , & \text{if}[\|\mathbf{a}^1\|^2 / \|\mathbf{p}\|^2 < \rho] \\ 0 , & \text{otherwise} \end{cases} .$$

For this problem

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}\|^2 = \left\| \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right\|^2 / \left\| \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\|^2 = \frac{2}{3} > \rho = 0.25 .$$

Therefore $a^0 = 0$, which agrees with the results of part (i).

P19.4 Show that the learning equation for the L2-L1 connections is equivalent to the outstar equation described in Chapter 15.

The L2-L1 learning law (Eq. (19.65)) is

$$\frac{d[\mathbf{w}_j^{2:1}(t)]}{dt} = a_j^2(t)[-\mathbf{w}_j^{2:1}(t) + \mathbf{a}^1(t)] .$$

If we approximate the derivative by

$$\frac{d[\mathbf{w}_j^{2:1}(t)]}{dt} \approx \frac{\mathbf{w}_j^{2:1}(t+\Delta t) - \mathbf{w}_j^{2:1}(t)}{\Delta t} ,$$

then we can rewrite Eq. (19.65) as

$$\mathbf{w}_j^{2:1}(t+\Delta t) = \mathbf{w}_j^{2:1}(t) + (\Delta t)a_j^2(t)\{-\mathbf{w}_j^{2:1}(t) + \mathbf{a}^1(t)\} .$$

This is equivalent to the outstar rule of Chapter 15 (Eq. (15.51)). Here the input to the L2-L1 connections is $a_j^2(t)$, and the output of the L2-L1 connections is $\mathbf{a}^1$.

P19.5 Train an ART1 network using the following input vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} .$$

Use the parameters $\zeta = 2$, and $\rho = 0.4$, and choose $S^2 = 3$ (3 categories).

Our initial weights will be

$$\mathbf{W}^{2:1} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \mathbf{W}^{1:2} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} .$$

We now begin the algorithm.

1. Compute the Layer 1 response:

$$\mathbf{a}^1 = \mathbf{p}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} .$$

2. Next, compute the input to Layer 2:

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}.$$

Since all neurons have the same input, pick the first neuron as the winner. (In case of a tie, pick the neuron with the smallest index.)

$$\mathbf{a}^2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

3. Now compute the L2-L1 expectation:

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \mathbf{w}_1^{2:1}.$$

4. Adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p}_1 \cap \mathbf{w}_1^{2:1} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \cap \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}_1\|^2 = \frac{1}{1} > \rho = 0.4, \text{ therefore } a^0 = 0 \text{ (no reset).}$$

6. Since $a^0 = 0$, continue with step 7.

7. Resonance has occurred, therefore update row 1 of $\mathbf{W}^{1:2}$:

$${}_1\mathbf{w}^{1:2} = \frac{2\mathbf{a}^1}{2 + \|\mathbf{a}^1\|^2 - 1} = \mathbf{a}^1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{W}^{1:2} = \begin{bmatrix} 0 & 1 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}.$$

8. Update column 1 of $\mathbf{W}^{2:1}$:

$$\mathbf{w}_1^{2:1} = \mathbf{a}^1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{W}^{2:1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

9. Remove $\mathbf{p}_1$, and return to step 1 with input pattern $\mathbf{p}_2$.

1. Compute the new Layer 1 response (Layer 2 inactive):

$$\mathbf{a}^1 = \mathbf{p}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

2. Next, compute the input to Layer 2:

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 0 & 1 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.5 \\ 0.5 \end{bmatrix}.$$

Since neurons 2 and 3 have the same input, pick the second neuron as the winner:

$$\mathbf{a}^2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}. \tag{19.76}$$

3. Now compute the L2-L1 expectation:

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \mathbf{w}_2^{2:1} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

4. Adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p}_2 \cap \mathbf{w}_2^{2:1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \cap \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}_2\|^2 = \frac{1}{1} > \rho = 0.4, \text{ therefore } a^0 = 0 \text{ (no reset).}$$

6. Since $a^0 = 0$, continue with step 7.

7. Resonance has occurred, therefore update row 2 of $\mathbf{W}^{1:2}$:

$$_2\mathbf{w}^{1:2} = \frac{2\mathbf{a}^1}{2 + \|\mathbf{a}^1\|^2 - 1} = \mathbf{a}^1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{W}^{1:2} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}.$$

8. Update column 2 of $\mathbf{W}^{2:1}$:

$$\mathbf{w}_2^{2:1} = \mathbf{a}^1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{W}^{2:1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

9. Remove $\mathbf{p}_2$, and return to step 1 with input pattern $\mathbf{p}_3$.

1. Compute the Layer 1 response with the new input vector:

$$\mathbf{a}^1 = \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

2. Next, compute the input to Layer 2:

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Since all neurons have the same input, pick the first neuron as the winner:

$$\mathbf{a}^2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

3. Now compute the L2-L1 expectation:

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{w}_1^{2:1} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

4. Adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p}_3 \cap \mathbf{w}_1^{2:1} = \begin{bmatrix}1\\1\\0\end{bmatrix} \cap \begin{bmatrix}0\\1\\0\end{bmatrix} = \begin{bmatrix}0\\1\\0\end{bmatrix}.$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}_3\|^2 = \frac{1}{2} > \rho = 0.4, \text{ therefore } a^0 = 0 \text{ (no reset).}$$

6. Since $a^0 = 0$, continue with step 7.

7. Resonance has occurred, therefore update row 1 of $\mathbf{W}^{1:2}$:

$${}_1\mathbf{w}^{1:2} = \frac{2\mathbf{a}^1}{2 + \|\mathbf{a}^1\|^2 - 1} = \mathbf{a}^1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \mathbf{W}^{1:2} = \begin{bmatrix}0 & 1 & 0\\1 & 0 & 0\\0.5 & 0.5 & 0.5\end{bmatrix}.$$

8. Update column 1 of $\mathbf{W}^{2:1}$:

$$\mathbf{w}_2^{2:1} = \mathbf{a}^1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \mathbf{W}^{2:1} = \begin{bmatrix}0 & 1 & 1\\1 & 0 & 1\\0 & 0 & 1\end{bmatrix}.$$

This completes the training, since if you apply any of the three patterns again they will not change the weights. These patterns have been successfully clustered. This type of result (stable learning) is guaranteed for the ART1 algorithm, since it has been proven to always produce stable clusters.

P19.6 Repeat Problem P19.5, but change the vigilance parameter to $\rho = 0.6$.

The training will proceed exactly as in Problem P19.5, until pattern $\mathbf{p}_3$ is presented, so let's pick up the algorithm at that point.

1. Compute the Layer 1 response:

$$\mathbf{a}^1 = \mathbf{p}_3 = \begin{bmatrix}1\\1\\0\end{bmatrix}.$$

2. Next, compute the input to Layer 2:

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Since all neurons have the same input, pick the first neuron as the winner:

$$\mathbf{a}^2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

3. Now compute the L2-L1 expectation:

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{w}_1^{2:1} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

4. Adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p}_3 \cap \mathbf{w}_1^{2:1} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \cap \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}_3\|^2 = \frac{1}{2} < \rho = 0.6, \text{ therefore } a^0 = 1 \text{ (reset).}$$

6. Since $a^0 = 1$, set $a_1^2 = 0$, inhibit it until an adequate match occurs (resonance), and return to step 1.

1. Recompute the Layer 1 response (Layer 2 inactive):

$$\mathbf{a}^1 = \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

2. Next, compute the input to Layer 2:

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Since neuron 1 is inhibited, neuron 2 is the winner:

$$\mathbf{a}^2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

3. Now compute the L2-L1 expectation:

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \mathbf{w}_2^{2:1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

4. Adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p}_3 \cap \mathbf{w}_2^{2:1} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \cap \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}_3\|^2 = \frac{1}{2} < \rho = 0.6, \text{ therefore } a^0 = 1 \text{ (reset).}$$

6. Since $a^0 = 1$, set $a_2^2 = 0$, inhibit it until an adequate match occurs (resonance), and return to step 1.

1. Recompute the Layer 1 response:

$$\mathbf{a}^1 = \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

2. Next, compute the input to Layer 2:

$$\mathbf{W}^{1:2}\mathbf{a}^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Since neurons 1 and 2 are inhibited, neuron 3 is the winner:

$$\mathbf{a}^2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

3. Now compute the L2-L1 expectation:

$$\mathbf{W}^{2:1}\mathbf{a}^2 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \mathbf{w}_3^{2:1} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

4. Adjust the Layer 1 output to include the L2-L1 expectation:

$$\mathbf{a}^1 = \mathbf{p}_3 \cap \mathbf{w}_3^{2:1} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \cap \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

5. Next, the Orienting Subsystem determines the degree of match between the expectation and the input pattern:

$$\|\mathbf{a}^1\|^2 / \|\mathbf{p}_3\|^2 = \frac{2}{2} > \rho = 0.6, \text{ therefore } a^0 = 0 \text{ (no reset).}$$

6. Since $a^0 = 0$, continue with step 7.

7. Resonance has occurred, therefore update row 3 of $\mathbf{W}^{1:2}$:

$${}_3\mathbf{w}^{1:2} = \frac{2\mathbf{a}^1}{2 + \|\mathbf{a}^1\|^2 - 1} = \frac{2}{3}\mathbf{a}^1 = \begin{bmatrix} \frac{2}{3} \\ \frac{2}{3} \\ 0 \end{bmatrix}, \quad \mathbf{W}^{1:2} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ \frac{2}{3} & \frac{2}{3} & 0 \end{bmatrix}.$$

8. Update column 3 of $\mathbf{W}^{2:1}$:

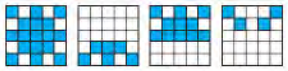
$$\mathbf{w}_3^{2:1} = \mathbf{a}^1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{W}^{2:1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

This completes the training, since if you apply any of the three patterns again they will not change the weights. (Verify this for yourself by applying

each input pattern to the network.) These patterns have been successfully clustered.

Note that in Problem P19.5, where the vigilance was $\rho = 0.4$, the patterns were clustered into two categories. In this problem, with vigilance $\rho = 0.6$, the patterns were clustered into three categories. The closer the vigilance is to 1, the more categories will be used. This is because an input pattern must be closer to a prototype in order to be incorporated into that prototype. When the vigilance is close to zero, many different input patterns can be incorporated into one prototype. The vigilance parameter adjusts the coarseness of the categorization.

P19.7 Train an ART1 network using the following input vectors (see [CaGr87a]):



$\mathbf{p}_1$ $\mathbf{p}_2$ $\mathbf{p}_3$ $\mathbf{p}_4$

Present the vectors in the order $\mathbf{p}_1 - \mathbf{p}_2 - \mathbf{p}_3 - \mathbf{p}_1 - \mathbf{p}_4$ (i.e., $\mathbf{p}_1$ is presented twice in each epoch). Use the parameters $\zeta = 2$ and $\rho = 0.6$, and choose $S^2 = 3$ (3 categories). Train the network until the weights have converged.

We begin by initializing the weight matrices. The initial $\mathbf{W}^{2:1}$ matrix is an $S^1 \times S^2 = 25 \times 3$ matrix of 1's. The initial $\mathbf{W}^{1:2}$ matrix is normalized, therefore it is an $S^2 \times S^1 = 3 \times 25$ matrix, with each element equal to

$$\frac{\zeta}{(\zeta + S^1 - 1)} = \frac{2}{(2 + 25 - 1)} = 0.0769 .$$

To create the input vectors we will scan each pattern row-by-row, where each blue square will be represented by a 1 and each white square will be represented by a 0. Since the input patterns are 5×5 grids, this will create 25-dimensional input vectors.

We now begin the training. Since it is not practical to display all of the calculations when the vectors are so large, we have summarized the results of the algorithm in Figure P19.4. Each row represents one iteration of the ART1 algorithm (presentation of one input vector). The left-most pattern in each row is the input vector. The remainder of the patterns represent the three columns of the $\mathbf{W}^{2:1}$ matrix. At each iteration, a star indicates the resonance point — the column of $\mathbf{W}^{2:1}$ that matched with the input pattern. Whenever a reset occurred, it is represented by a check mark. When more than one reset occurred in a given iteration, the number beside the check mark indicates the order in which the reset occurred.

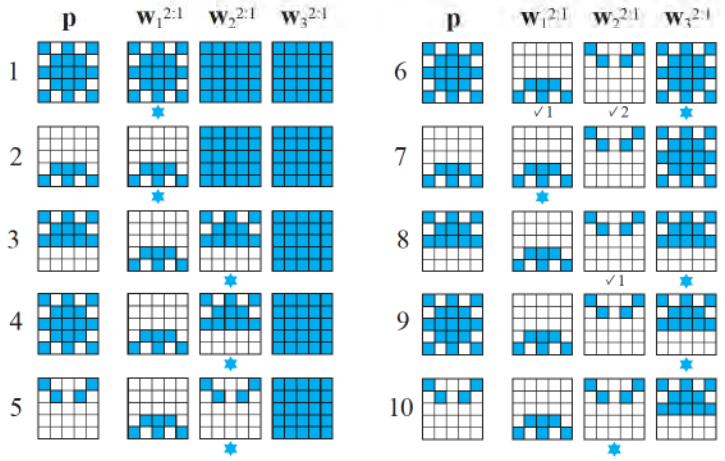


Figure P19.4 ART1 Iterations for Problem

A total of 10 iterations of the algorithm were performed (two epochs of the sequence $\mathbf{p}_1 - \mathbf{p}_2 - \mathbf{p}_3 - \mathbf{p}_1 - \mathbf{p}_4$). The weights are now stable. (You may want to check this by presenting each input pattern.)

There are several interesting points to notice in this example. First, notice that at iteration 4 both $\mathbf{p}_1$ and $\mathbf{p}_3$ are coded by $\mathbf{w}_2^{2:1}$. However, on iteration 5, when $\mathbf{p}_4$ is presented, $\mathbf{w}_2^{2:1}$ is modified to include $\mathbf{p}_4$. This new $\mathbf{w}_2^{2:1}$ no longer provides an adequate match with $\mathbf{p}_1$ and $\mathbf{p}_3$, as we can see at iterations 6 and 8. This requires them to take over neuron 3, which was unused during the first epoch.

The results of the algorithm could be modified by changing the vigilance parameter. How small would you have to make the vigilance, so that only two neurons in Layer 2 would be required to code all 4 input vectors? How large would the vigilance have to be before a fourth Layer 2 neuron was needed?

Epilogue

Competitive learning, and many other types of neural network training algorithms, suffer from a problem called the stability/plasticity dilemma. If a learning algorithm is sensitive to new inputs (plastic), then it runs the risk of forgetting prior learning (unstable). The ART networks were designed to achieve learning stability while maintaining sensitivity to novel inputs.

In this chapter, the ART1 network was used to illustrate the key concepts of adaptive resonance theory. The ART1 network is based on the Grossberg competitive network of Chapter 18, with a few modifications. The key innovation of ART is the use of "expectations." As each input pattern is presented to the network, it is compared with the prototype vector that it most closely matches (the expectation). If the match between the prototype and the input vector is not adequate, a new prototype is selected. In this way, previously learned memories (prototypes) are not eroded by new learning.

One important point to keep in mind when analyzing ART networks, is that they were designed to be biologically plausible mechanisms for learning. They have as much to do with understanding how the brain works as they do with inspiring practical pattern recognition systems. For this reason, the learning mechanisms are required to use only local information at each neuron. This is not true of all of the learning rules discussed in this text.

Although the ART networks solve the problem of learning instability, in which the network weights never stabilize, there is another stability problem that we have not yet discussed. This is the stability of the differential equations that implement the short-term memory equations of the network. In Layer 2, for example, we have a set of differential equations with nonlinear feedback. Can we make some general statement about the stability of such systems? Chapter 20 will present a comprehensive discussion of this problem.

Further Reading

[CaGr87a] G. A. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics, and Image Processing*, vol. 37, pp. 54–115, 1987.

In this original presentation of the ART1 architecture, Carpenter and Grossberg demonstrate that the architecture self-organizes and self-stabilizes in response to an arbitrary number of binary input patterns. The key feature of ART is a top-down matching mechanism.

[CaGr87b] G. A. Carpenter and S. Grossberg, "ART2: Self-organization of stable category recognition codes for analog input patterns," *Applied Optics*, vol. 26, no. 23, pp. 4919–4930, 1987.

This article describes an extension of the ART1 architecture that is designed to handle analog input patterns.

[CaGr90] G. A. Carpenter and S. Grossberg, "ART3: Hierarchical search using chemical transmitters in self-organizing pattern recognition architectures," *Neural Networks*, vol. 3, no. 23, pp. 129–152, 1990.

This article demonstrates how the Orienting Subsystem of the ART networks could be implemented in biological neurons through the use of chemical transmitters.

[CaGrMa92] G. A. Carpenter, S. Grossberg, N. Markuzon, J. Reynolds and D. Rosen, "Fuzzy ARTMAP: An adaptive resonance architecture for incremental learning of analog maps," *Proceedings of the International Joint Conference on Neural Networks*, Baltimore, MD, vol. 3, no. 5, pp. 309–314, 1992.

The authors present a modification of the ARTMAP architecture to include fuzzy logic that enables better performance in a noisy environment.

[CaGrRe91] G. A. Carpenter, S. Grossberg and J. Reynolds, "ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network," *Neural Networks*, vol. 4, no. 5, pp. 169–181, 1991.

This article presents an adaptive resonance theory network for supervised learning. The network consists of two interconnected ART modules. One module receives the input vector, and the other module receives the desired output vector.

[Gros76] S. Grossberg, "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors," *Biological Cybernetics*, vol. 23, pp. 121–134, 1976.

Grossberg describes a continuous-time competitive network, inspired by the developmental physiology of the visual cortex. The structure of this network forms the foundation for other important networks.

[Gros82] S. Grossberg, *Studies of Mind and Brain*, Boston: D. Reidel Publishing Co., 1982.

This book is a collection of Stephen Grossberg papers from the period 1968 through 1980. It covers many of the fundamental concepts used in later Grossberg networks, such as the adaptive resonance theory networks.

Exercises

E19.1 Consider Layer 1 of the ART1 network with $\varepsilon = 0.02$. Assume two neurons in Layer 2, two elements in the input vector and the following weight matrix and input:

$$\mathbf{W}^{2:1} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Also assume that neuron 2 of Layer 2 is active.

i. Find and plot the response $\mathbf{n}^1$ if ${}^{+}b^1 = 2$ and ${}^{-}b^1 = 3$.

ii. Find and plot the response $\mathbf{n}^1$ if ${}^{+}b^1 = 4$ and ${}^{-}b^1 = 5$.

iii. Find and plot the response $\mathbf{n}^1$ if ${}^{+}b^1 = 4$ and ${}^{-}b^1 = 4$.

iv. Check to see that the answers to parts (i)–(iii) satisfy the steady state response predicted by Eq. (19.21). Explain any inconsistencies.



v. Check your answers to parts (i)–(iii) by writing a MATLAB M-file to simulate Layer 1 of the ART1 network. Use the `ode45` routine. Plot the response for each case.

E19.2 Consider Layer 2 of the ART1 network with the following parameters:

$$\varepsilon = 0.1 \quad \mathbf{W}^{1:2} = \begin{bmatrix} ({}_1\mathbf{w}^{1:2})^T \\ ({}_2\mathbf{w}^{1:2})^T \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & \frac{2}{3} \\ 1 & 0 \end{bmatrix}$$

and

$$f^2(n) = \begin{cases} 10(n)^2, & n \geq 0 \\ 0, & n < 0 \end{cases}.$$

Assume that the output of Layer 1 is

$$\mathbf{a}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$



i. Write the equations of operation of Layer 2, and simulate and plot the response if the following bias vectors are used:

$$^{+}\mathbf{b}^{2} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad ^{-}\mathbf{b}^{2} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}.$$

ii. Repeat part (i) for the following bias vectors:

$$^{+}\mathbf{b}^{2} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \quad ^{-}\mathbf{b}^{2} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}.$$

iii. Repeat part (i) for the following bias vectors:

$$^{+}\mathbf{b}^{2} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \quad ^{-}\mathbf{b}^{2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

iv. Do the results of all of the previous parts satisfy the desired steady state response described in Eq. (19.37)? If not, explain why.

E19.3 Consider the Orienting Subsystem of the ART1 network with the following parameters:

$$\varepsilon = 0.1 \quad ^{+}b^{0} = ^{-}b^{0} = 2.$$

The inputs to the Orienting Subsystem are

$$\mathbf{p} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{a}^{1} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

i. Find and plot the response of the Orienting Subsystem $n^{0}(t)$, for $\alpha = 0.5 \quad \beta = 4 \quad (\rho = 0.125)$.

ii. Find and plot the response of the Orienting Subsystem $n^{0}(t)$, for $\alpha = 0.5 \quad \beta = 2 \quad (\rho = 0.25)$.

iii. Verify that the steady state conditions are satisfied in parts (i) and (ii).



iv. Check your answers to parts (i) and (ii) by writing a MATLAB M-file to simulate the Orienting Subsystem.

E19.4 To derive the steady state conditions for the L1-L2 and L2-L1 learning rules, we have made the assumption that the input pattern and the neuron outputs remain constant until the weight matrices converge. This is called "fast learning." Show that this fast learning assumption is equivalent to setting the learning rate α to 1 in the instar and outstar learning rules presented in Chapter 15 and the Kohonen competitive learning rule in Chapter 16.

E19.5 Train an ART1 network using the following input vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Use the parameter $\zeta = 2$, and choose $S^2 = 3$ (3 categories).

i. Train the network to convergence using $\rho = 0.3$.

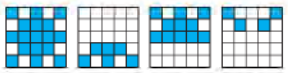
ii. Repeat part (i) using $\rho = 0.6$.

iii. Repeat part (ii) using $\rho = 0.9$.

E19.6 The ART1 algorithm can be modified to add a new neuron in Layer 2 whenever there is no adequate match between the existing prototypes and the input pattern. This involves creating a new row of the $\mathbf{W}^{1:2}$ matrix and a new column of the $\mathbf{W}^{2:1}$ matrix. Describe how this would be done.

E19.7 Write a Matlab M-file to implement the ART1 algorithm (with the modification described in Exercise E19.6). Use this M-file to train an ART1 network using the following input vectors (see Problem P19.7):



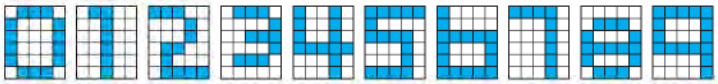


$\mathbf{p}_1$ $\mathbf{p}_2$ $\mathbf{p}_3$ $\mathbf{p}_4$

Present the vectors in the order $\mathbf{p}_1 - \mathbf{p}_2 - \mathbf{p}_3 - \mathbf{p}_1 - \mathbf{p}_4$ (i.e., $\mathbf{p}_1$ is presented twice in each epoch). Use the parameters $\zeta = 2$ and $\rho = 0.9$, and choose $S^2 = 3$ (3 categories). Train the network until the weights have converged. Compare your results with Problem P19.7.

E19.8 Recall the digit recognition problem described in Chapter 7 (page 7-10). Train an ART1 network using the digits $0-9$, as displayed below:





Use the parameter $\zeta = 2$, and choose $S^2 = 5$ (5 categories). Use the Matlab M-file from Exercise E19.7.

i. Train the network to convergence using $\rho = 0.3$.

ii. Train the network to convergence using $\rho = 0.6$.

iii. Train the network to convergence using $\rho = 0.9$.

iv. Discuss the results of parts (i)–(iii). Explain the effect of the vigilance parameter.

20 Stability

Objectives	20-1
Theory and Examples	20-2
Recurrent Networks	20-2
Stability Concepts	20-3
Definitions	20-4
Lyapunov Stability Theorem	20-5
Pendulum Example	20-6
LaSalle's Invariance Theorem	20-12
Definitions	20-12
Theorem	20-13
Example	20-14
Comments	20-18
Summary of Results	20-19
Solved Problems	20-21
Epilogue	20-28
Further Reading	20-29
Exercises	20-30

Objectives

The problem of "convergence" in a recurrent network was first raised in our discussion of the Hopfield network, in Chapter 3. It was noted there that the output of a recurrent network could converge to a stable point, oscillate, or perhaps even diverge. The "stability" of the steepest descent process and of the LMS algorithm were discussed in Chapter 9 and Chapter 10, respectively. The stability of Grossberg's continuous-time recurrent networks was discussed in Chapter 18.

In this chapter we will define stability more carefully. Our objective is to determine whether a particular set of nonlinear equations has points (or trajectories) to which its output might converge. To help us study this topic we will introduce Lyapunov's Stability Theorem and apply it to a simple, but instructive, problem. Then, we will present a generalization of the Lyapunov Theory: LaSalle's Invariance Theorem. This will set the stage for Chapter 21, where LaSalle's theorem is used to prove the stability of Hopfield networks.

Theory and Examples

Recurrent Networks

We first discussed recurrent neural networks, which have feedback connections from their outputs to their inputs, when we introduced the Hamming and Hopfield networks in Chapter 3. The Grossberg networks of Chapter 18 and Chapter 19 also contain recurrent connections. Recurrent networks are potentially more powerful than feedforward networks, since they are able to recognize and recall temporal, as well as spatial, patterns. However, the behavior of these recurrent networks is much more complex than that of feedforward networks.

For feedforward networks, the output is constant (for a fixed input) and is a function only of the network input. For recurrent networks, however, the output of the network is a function of time. For a given input and a given initial network output, the response of the network may converge to a stable output. However, it may also oscillate, explode to infinity, or follow a chaotic pattern. In the remainder of this chapter we want to investigate general nonlinear recurrent networks, in order to determine their long-term behavior.

We will consider recurrent networks that can be described by nonlinear differential equations of the form:

$$\frac{d}{dt}\mathbf{a}(t) = \mathbf{g}(\mathbf{a}(t),\mathbf{p}(t),t). \tag{20.1}$$

Here $\mathbf{p}(t)$ is the input to the network, and $\mathbf{a}(t)$ is the output of the network. (See Figure 20.1.)

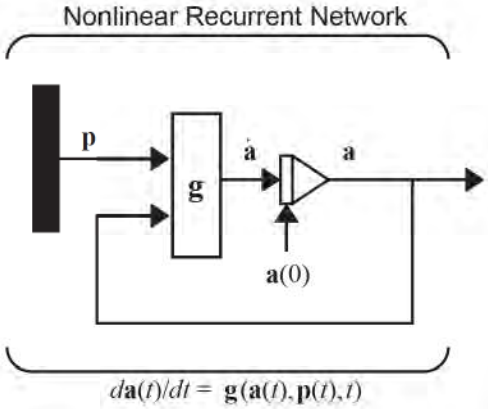


Figure 20.1 Nonlinear, Continuous-Time, Recurrent Network

We want to know how these systems perform in the steady state. We will be most interested in those cases where the network converges to a constant output — a stable equilibrium point. A nonlinear system can have many stable points. For some neural networks these stable points represent stored prototype patterns. When possible, we would like to know where the stable points are, and which initial conditions $\mathbf{a}(0)$ converge to a given stable point (i.e., what is the basin of attraction for a given stable point?).

Stability Concepts



To begin our discussion, let's introduce some basic stability concepts with a simple, intuitive example. Consider the motion of a ball bearing, with dissipative friction, in a gravity field. In the adjacent figure, we have a ball bearing at the bottom of a trough (point $\mathbf{a}^*$). If we move the bearing to a different position, it will oscillate back and forth in the trough, but, because of friction, it will eventually settle back to the bottom of the trough. We will call this position an *asymptotically stable* point, which we will define more precisely in the next section.



Consider now the second figure in the left margin. Here we have a ball bearing positioned at the center of a flat surface. If we place the bearing in a different position, it will not move. The position at the center of the surface is not asymptotically stable, since the bearing does not move back to the center if it is moved away. However, it is stable in a certain sense, because at least the ball does not roll farther away from the center point. We call this kind of point *stable in the sense of Lyapunov*, which we will define in the next section.



Now consider the third figure in the left margin. The ball bearing is positioned at the top of a hill. This is an equilibrium point, since the ball will remain at the top of the hill, if we position it carefully. However, if the bearing is given the slightest disturbance, it will roll down the hill. This is an *unstable* equilibrium point.

In the next chapter we will try to design Hopfield neural networks, in which the stored prototype patterns will be asymptotically stable equilibrium points. We would also like the basins of attraction for these stable points to be as large as possible.

For example, consider Figure 20.2. We would like to design neural networks with large basins of attraction such as those of Case A. One can certainly imagine that if a ball that rolls with high friction is placed (with zero velocity) in any one of the basins of Case A, it will remain in that basin and will eventually find its way to the bottom (stable point). However, Case B is more complicated. If, for instance, one places a ball with friction at point P, it is not clear which stable point will eventually capture the ball. The ball may not come to rest at the stable point closest to P. It is also difficult to tell how large the basin of attraction is for a specific stable point.

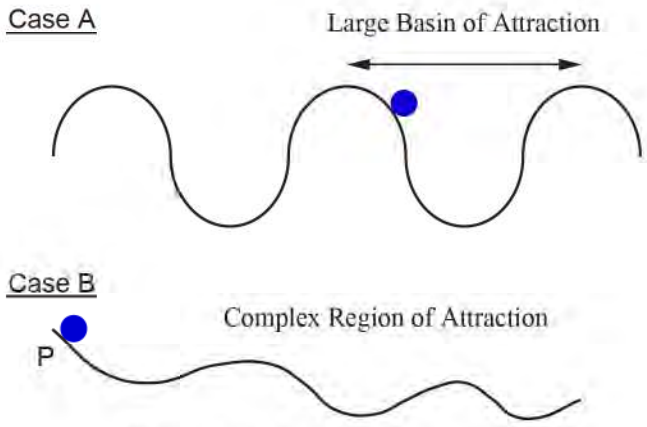


Figure 20.2 Basins of Attraction

Now that we have presented some intuitive notions of stability, we will pursue them with mathematical rigor in the remainder of this chapter.

Definitions

Equilibrium Point

We will begin with specific mathematical definitions of the different types of stability discussed in the previous section. In these definitions we will be talking about the stability of an *equilibrium point*; a point $\mathbf{a}^*$ where the derivative in Eq. (20.1) is zero. For simplicity, we will talk specifically about the point $\mathbf{a}^* = \mathbf{0}$, which is referred to as the origin. This restriction does not affect the generality of our discussion.

Stability

Definition 1: *Stability* (in the sense of Lyapunov)

The origin is a stable equilibrium point if for any given value $\varepsilon > 0$ there exists a number $\delta(\varepsilon) > 0$ such that if $\|\mathbf{a}(0)\| < \delta$, then the resulting motion $\mathbf{a}(t)$ satisfies $\|\mathbf{a}(t)\| < \varepsilon$ for $t > 0$.

This definition says that the system output is not going to move too far away from a given stable point, so long as it is initially close to the stable point. Let's say that you want the system output to remain within a distance ε of the origin. If the origin is stable, then you can always find a distance δ (which may be a function of ε), such that if the system output is within δ of the origin at time $t = 0$, then it will always remain within ε of the origin. The position of the ball (with zero velocity) in the figure to the left is stable in the sense of Lyapunov, so long as the ball bearing has friction. If the ball bearing did not have friction, then any initial velocity would produce a trajectory $\mathbf{a}(t)$ in which the position would go to infinity. (The vector $\mathbf{a}(t)$ in this case would consist of the position and the velocity of the ball.)



Next, let's consider the stronger concept of asymptotic stability.

Asymptotic Stability

Definition 2: *Asymptotic Stability*

The origin is an asymptotically stable equilibrium point if there exists a number $\delta > 0$ such that whenever $\|\mathbf{a}(0)\| < \delta$ the resulting motion satisfies $\|\mathbf{a}(t)\| \to 0$ as $t \to \infty$.



This is a stronger definition of stability. It says that as long as the output of the system is initially within some distance δ of the stable point, the output will eventually converge to the stable point. The position of the ball (with zero velocity) in the diagram in the left margin is asymptotically stable, so long as the ball bearing has friction. If there is no friction, the position is only stable in the sense of Lyapunov.

We would like to build neural networks that have many specified asymptotically stable points, each of which represents a prototype pattern. This is the design objective we will use for building Hopfield networks in Chapter 21.

In addition to the stability definitions, there is another concept we will use in analyzing stability. It is the concept of a *definite* function. The next two definitions will clarify this concept.

Positive Definite

Definition 3: *Positive Definite*

A scalar function $V(\mathbf{a})$ is positive definite if $V(\mathbf{0}) = 0$ and $V(\mathbf{a}) > 0$ for $\mathbf{a} \neq \mathbf{0}$.

Positive Semidefinite

Definition 4: *Positive Semidefinite*

A scalar function $V(\mathbf{a})$ is positive semidefinite if $V(\mathbf{a}) \geq 0$ for all $\mathbf{a}$.

(These definitions can be modified appropriately to define the concepts *negative definite* and *negative semidefinite*.) Now that we have defined stability, let's consider a method for testing stability.

Lyapunov Stability Theorem

One of the most important approaches for investigating the stability of nonlinear systems is the theory introduced by Alexandr Mikhailovich Lyapunov, a Russian mathematician. Although his major work was first published in 1892, it received little attention outside Russia until much later. In this section we will discuss one of Lyapunov's most powerful techniques for stability analysis — the so-called *direct method*.

Consider the autonomous (unforced, no explicit time dependence) system:

$$\frac{d\mathbf{a}}{dt} = \mathbf{g}(\mathbf{a}). \tag{20.2}$$

The Lyapunov stability theorem can then be stated as follows.

Theorem 1: *Lyapunov Stability Theorem*

If a positive definite function $V(\mathbf{a})$ can be found such that $dV(\mathbf{a})/dt$ is negative semidefinite, then the origin ($\mathbf{a} = \mathbf{0}$) is stable for the system of Eq. (20.2). If a positive definite function $V(\mathbf{a})$ can be found such that $dV(\mathbf{a})/dt$ is negative definite, then the origin ($\mathbf{a} = \mathbf{0}$) is asymptotically stable. In each case, V is called a Lyapunov function of the system.

You can think of $V(\mathbf{a})$ as a generalized energy function. The concept of the theorem is that if the energy of a system is continually decreasing ($dV(\mathbf{a})/dt$ negative definite), then it will eventually settle at some minimum energy state. Lyapunov's insight was to generalize the concept of energy, so that the theorem could be applied to systems where the energy is difficult to express or has no meaning.

We should note that the theorem only states that if a suitable Lyapunov function $V(\mathbf{a})$ can be found, the system is stable. It gives us no information about the stability of the system in those situations where we are unable to find such a function.

Pendulum Example

We can gain some insight into Lyapunov's stability theorem by applying it to a simple mechanical system. This system is very simple, and its operation is easy to visualize, and yet it illustrates important concepts that we will apply to neural network design in the next chapter. The example system we will use is the pendulum shown in Figure 20.3.

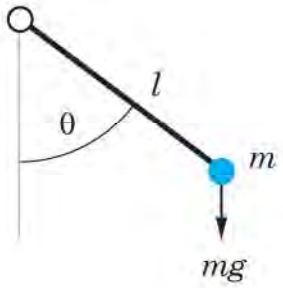


Figure 20.3 Pendulum

Using Newton's second law ($F = ma$), we can write the equation of operation of the pendulum as

$$ml\frac{d^2}{dt^2}(\theta) = -c\frac{d\theta}{dt} - mg\sin(\theta), \tag{20.3}$$

or

$$ml\frac{d^2\theta}{dt^2} + c\frac{d\theta}{dt} + mg\sin(\theta) = 0, \tag{20.4}$$

where θ is the angle of the pendulum, m is the mass of the pendulum, l is the length of the pendulum, c is the damping coefficient, and g is the gravitational constant.

The first term on the right side of Eq. (20.3) is the damping force, which is proportional to the velocity of the pendulum. It is this term that represents the energy dissipation in the system. The second term on the right side of Eq. (20.3) is the gravitational force, which is proportional to the sine of the angle of the pendulum. It is equal to zero when the pendulum is straight down and has its maximum value when the pendulum is horizontal.

If the damping coefficient is not zero, the pendulum will eventually come to rest hanging down in the vertical position. This solution might be viewed as $\theta = 0$, but more generally it is $\theta = 2\pi n$, where $n = 0, \pm 1, \pm 2, \pm 3, \ldots$. That is, given the appropriate initial conditions, the pendulum might simply settle to $\theta = 0$ or it might rotate once to give a solution of $\theta = 2\pi$, etc. There are many possible equilibrium solutions. (The positions $\theta = \pi n$, for odd values of n, are also equilibrium points, but they are not stable.)

To analyze the stability of this system, we will write the pendulum equation in state variable form, where it will appear as a pair of first-order differential equations. Let's choose the following state variables:

$$a_1 = \theta \text{ and } a_2 = \frac{d\theta}{dt}. \tag{20.5}$$

We can write equations for the pendulum in terms of these state variables as follows:

$$\frac{da_1}{dt} = a_2, \tag{20.6}$$

$$\frac{da_2}{dt} = -\frac{g}{l}\sin(a_1) - \frac{c}{ml}a_2. \tag{20.7}$$

Now we want to investigate the stability of the origin ($\mathbf{a} = \mathbf{0}$) for this pendulum system. (The origin corresponds to a pendulum angle of zero and a pendulum velocity of zero.) We first want to check that the origin is an equilibrium point. We do this by substituting $\mathbf{a} = \mathbf{0}$ into the state equations.

$$\frac{da_1}{dt} = a_2 = 0, \tag{20.8}$$

$$\frac{da_2}{dt} = -\frac{g}{l}\sin(a_1) - \frac{c}{ml}a_2 = -\frac{g}{l}\sin(0) - \frac{c}{ml}(0) = 0 \tag{20.9}$$

Since the derivatives are zero, the origin is an equilibrium point.

Next we need to find a Lyapunov function for the pendulum. For this example we will use the energy of the system as the Lyapunov function V. To obtain the total energy of the pendulum, we add the kinetic and potential energies.

$$V(\mathbf{a}) = \frac{1}{2}ml^2(a_2)^2 + mgl(1 - \cos(a_1)) \tag{20.10}$$

In order to test the stability of the system, we need to evaluate the derivative of V with respect to time.

$$\frac{d}{dt}V(\mathbf{a}) = [\nabla V(\mathbf{a})]^T \mathbf{g}(\mathbf{a}) = \frac{\partial V}{\partial a_1}\left(\frac{da_1}{dt}\right) + \frac{\partial V}{\partial a_2}\left(\frac{da_2}{dt}\right) \tag{20.11}$$

The partial derivatives of $V(\mathbf{a})$ can be obtained from Eq. (20.10), and the derivatives of the two state variables are given by Eq. (20.6) and Eq. (20.7). Thus we have

$$\frac{d}{dt}V(\mathbf{a}) = (mgl\sin(a_1))a_2 + (ml^2a_2)\left(-\frac{g}{l}\sin(a_1) - \frac{c}{ml}a_2\right). \tag{20.12}$$

The $(mgl\sin(a_1))a_2$ terms cancel, which leaves only

$$\frac{d}{dt}V(\mathbf{a}) = -cl(a_2)^2 \le 0. \tag{20.13}$$

In order to prove that the origin ($\mathbf{a} = \mathbf{0}$) is asymptotically stable, we must show that this derivative is negative definite. The derivative is zero at the origin, but it also is zero for any value of a_1, as long as $a_2 = 0$. Thus, $dV(\mathbf{a})/\mathrm{dt}$ is negative semidefinite, rather than negative definite. From Lyapunov's theorem, then, we know that the origin is a stable point. However, we *cannot* say, from the theorem and this Lyapunov function, that the origin is asymptotically stable.

In this case we know that as long as the pendulum has friction, it will eventually settle in a vertical position, and, therefore, that the origin is asymptotically stable. However, Lyapunov's theorem, using our Lyapunov function, can only tell us that the origin is stable. To prove that the origin is asymptotically stable, we will need a refinement of Lyapunov's theorem, LaSalle's Invariance Theorem. We will discuss LaSalle's theorem in the next section.

First, let's investigate the pendulum further, by taking a specific numerical example. Let $g = 9.8$, $m = 1$, $l = 9.8$, $c = 1.96$. Now we can rewrite the state equations for the pendulum as

$$\frac{da_1}{dt} = a_2, \tag{20.14}$$

$$\frac{da_2}{dt} = -\sin(a_1) - 0.2a_2. \tag{20.15}$$

Expressions for V and its derivative follow:

$$V = (9.8)^2\left[\frac{1}{2}(a_2)^2 + (1 - \cos(a_1))\right], \tag{20.16}$$

$$\frac{dV}{dt} = -(19.208)(a_2)^2. \tag{20.17}$$

Note that dV/dt is zero for any value of a_1 as long as $a_2 = 0$.

Figure 20.4 displays the 3-D and contour plots of the energy surface, V, as the angle varies between -10 and +10 radians and the angular velocity varies between -2 and 2 radians per second. Note that in this range there are three possible minimum points of the energy surface, at 0 and $\pm 2\pi$.

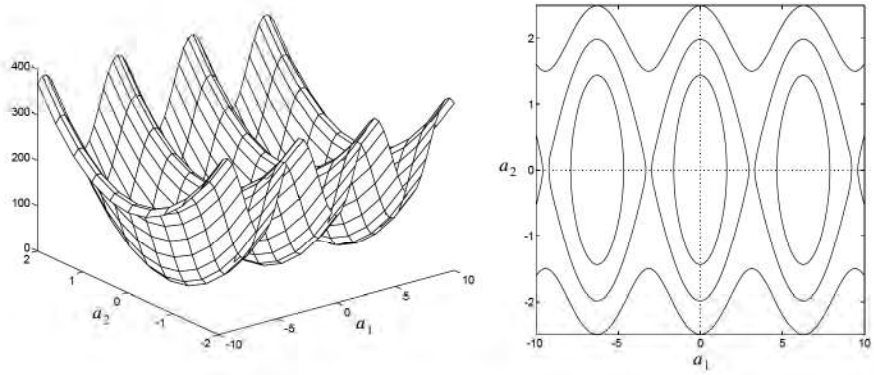


Figure 20.4 Pendulum Energy Surface

(We will find in Chapter 21 that the minimum points of the Lyapunov function can correspond to prototype patterns in an autoassociative neural network. The pendulum system, like recurrent neural networks, has many minimum points.)

Of course, the energy plots shown in Figure 20.4 do not tell us in what way, or by what route, the pendulum finds a particular energy minimum. To

show this, we have plotted the energy contours, and one particular path for the pendulum, in Figure 20.5. The response trajectory, shown by the blue line, starts from an initial position, $a_1(0)$, of 1.3 radians (74°) and an initial velocity, $a_2(0)$, of 1.3 radians per second. The trajectory converges to the equilibrium point $\mathbf{a} = \mathbf{0}$.

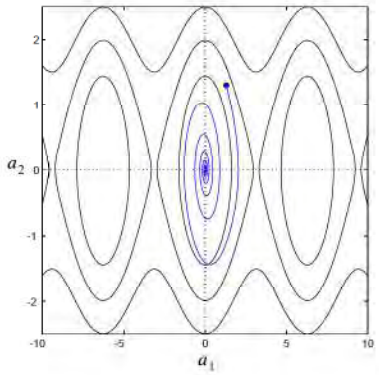


Figure 20.5 Pendulum Response on State Variable Plane

A time response plot of the two state variables is shown in Figure 20.6. Notice that, because the initial velocity is positive, the pendulum continues to move up initially. (Check to see if this agrees with Figure 20.5.) It reaches a maximum angle of about 2 radians before falling back down. The oscillations continue to decay as both state variables converge to zero.

In this case, both state variables converge to zero. However, this is not the only possible equilibrium point, as we will show later.

It is also interesting to plot the pendulum energy, V, as in Figure 20.7. Recall from Eq. (20.17) that the energy should never increase; this is consistent with Figure 20.7. Eq. (20.17) also predicts that the derivative of the energy curve should only be zero when the velocity, a_2, is zero. This is also verified if we compare Figure 20.7 with Figure 20.6. At those times where the a_2 graph crosses the zero axis, the slope of the energy curve is zero.

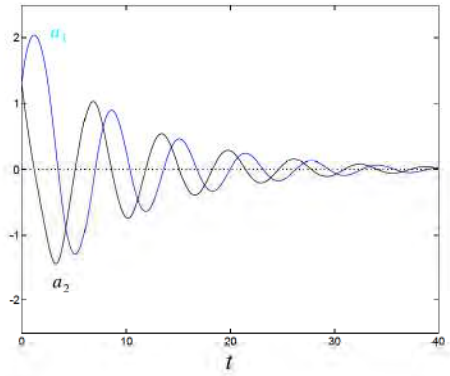


Figure 20.6 State Variables a_1 (blue) and a_2 vs. Time

Notice that, although there are points where the derivative of the energy curve is zero, the derivative does not remain zero until the energy is also zero. This observation will lead to LaSalle's Invariance Theorem, which we will discuss in the next section. The key idea of that theorem is to identify those points where the derivative of the Lyapunov function is zero, and then to determine if the system will be trapped at those points. (Those places where a trajectory can be trapped are called invariant sets.) If the only point that can trap the trajectory, and that has zero derivative, is the origin, then the origin is asymptotically stable.

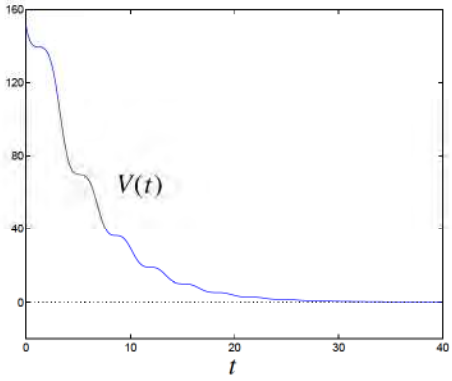


Figure 20.7 Pendulum Lyapunov Function (Energy) vs. Time

The particular pendulum behavior shown in the graphs in this section depends on the initial conditions of the two state variables. The choice of a different set of initial conditions may give results entirely different from those shown in these plots. We will expand on this in the next section.



To experiment with the pendulum, use the Neural Network Design Demonstration Dynamic System (`nnd17ds`).

LaSalle's Invariance Theorem

The pendulum example demonstrated a problem with Lyapunov's theorem. We found a Lyapunov function whose derivative was only negative semidefinite (not negative definite), and yet we know that the origin is asymptotically stable for the pendulum system. In this section we will introduce a theorem that clarifies this uncertainty in Lyapunov's theorem. It does so by defining those regions of the state space where the derivative of the Lyapunov function is zero, and then identifying those parts of that region that can trap the trajectory.

Before we discuss LaSalle's Invariance Theorem, we first need to introduce the following definitions.

Definitions

Lyapunov Function

Definition 5: *Lyapunov Function*

Let V be a continuously differentiable function from $\Re^n$ to $\Re$. If G is any subset of $\Re^n$, we say that V is a Lyapunov function on G for the system $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$ if

$$\frac{dV(\mathbf{a})}{dt} = (\nabla V(\mathbf{a}))^T \mathbf{g}(\mathbf{a}) \tag{20.18}$$

does not change sign on G.

This is a generalization of our previous definition of the Lyapunov function, which we used in Theorem 1. Here we do not require that the function be positive definite. In fact, there is no direct requirement on the function itself (except that it be continuously differentiable). The only requirement is on the derivative of V. The derivative cannot change sign anywhere on the set G. Note that the derivative will not change sign if it is negative semidefinite or if it is positive semidefinite.

We should note here that we have not yet explained how to choose the set G. We will use the following definitions and theorems to help us select the best G for a given system.

Set Z

Definition 6: *Set* Z

$$Z = \{\mathbf{a}: dV(\mathbf{a})/dt = 0, \ \mathbf{a} \text{ in the closure of } G\}. \tag{20.19}$$

Here "the closure of G" includes the interior and the boundary of G. This is a key set. It contains all of those points where the derivative of the Lyapunov function is zero. Later we will want to determine where in this set the system trajectory can be trapped.

Invariant Set

Definition 7: *Invariant Set*

A set of points in $\Re^n$ is *invariant* with respect to $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$ if every solution of $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$ starting in that set remains in the set for all time.

If the system gets into an invariant set, then it can't get out.

Set L

Definition 8: *Set* L

L is defined as the largest invariant set in Z.

This set includes all possible points at which the solution might converge. The Lyapunov function does not change in L (because its derivative is zero), and the trajectory will be trapped in L (because it is an invariant set). Now, if this set has only one stable point, then that point is asymptotically stable. This is, in essence, what LaSalle's theorem will say.

Theorem

LaSalle's Invariance Theorem extends the Lyapunov Stability Theorem. We will use it to design Hopfield networks in the next chapter. The theorem proceeds as follows [Lasa67].

Theorem 2: *LaSalle's Invariance Theorem*

If V is a Lyapunov function on G for $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$, then each solution $\mathbf{a}(t)$ that remains in G for all $t > 0$ approaches $L^\circ = L \cup \{\infty\}$ as $t \to \infty$. (G is a basin of attraction for L, which has all of the stable points.) If all trajectories are bounded, then $\mathbf{a}(t) \to L$ as $t \to \infty$.

If a trajectory stays in G, then it will either converge to L, or it will go to infinity. If all trajectories are bounded, then all trajectories will converge to L.

There is a corollary to LaSalle's theorem that we will use extensively. It involves choosing the set G in a special way.

Corollary 1: *LaSalle's Corollary*

Let G be a component (one connected subset) of

$$\Omega_\eta = \{\mathbf{a}:V(\mathbf{a}) < \eta\}. \tag{20.20}$$

Assume that G is bounded, $dV(\mathbf{a})/dt \le 0$ on the set G, and let the set $L^\circ = closure(L \cap G)$ be a subset of G. Then L° is an attractor, and G is in its region of attraction.

LaSalle's theorem, and its corollary, are very powerful. Not only can they tell us which points are stable (L°), but they can also provide us with a partial region of attraction (G). (Note that L° is constructed differently in the corollary than in the theorem.)

To clarify LaSalle's Invariance Theorem, let's return to the pendulum example we discussed earlier.

Example



Let's apply Corollary 1 to the pendulum example. The first step in using the corollary will be to choose the set Ω_η. This set will then be used to select the set G (a component of Ω_η).

For this example we will use the value $\eta = 100$, therefore Ω_{100} will be the set of points where the energy is less than or equal to 100.

$$\Omega_{100} = \{\mathbf{a}:V(\mathbf{a}) \le 100\} \tag{20.21}$$

This set is displayed in blue in Figure 20.8.

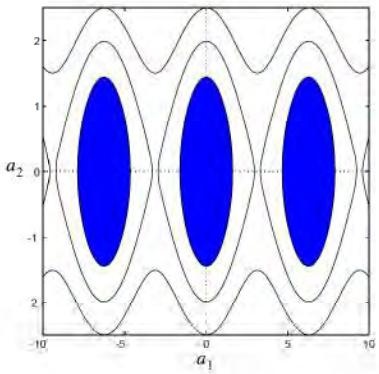


Figure 20.8 Illustration of the Set Ω_{100}

The next step in our analysis is to choose a component (connected subset) of Ω_{100} for the set G. Since we have been investigating the stability of the

origin, let's choose the component of Ω_{100} that contains $\mathbf{a} = \mathbf{0}$. The resulting set is shown in Figure 20.9.

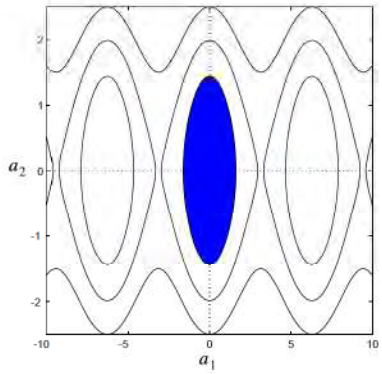


Figure 20.9 Illustration of the Set G

Now that we have chosen G, we need to check that the derivative of the Lyapunov function is less than or equal to zero on G. From Eq. (20.17) we know that $dV(\mathbf{a})/dt$ is negative semidefinite. Therefore it will certainly be less than or equal to zero on G.

We are now ready to determine the attractor set L°. We begin with the set L, which is the largest invariant set in Z.

$$\begin{aligned} Z &= \{\mathbf{a}\colon dV(\mathbf{a})/dt = 0,\ \mathbf{a} \text{ in the closure of } G\} \\ &= \{\mathbf{a}\colon a_2 = 0,\ \mathbf{a} \text{ in the closure of } G\}\ . \end{aligned} \tag{20.22}$$

This can also be written as

$$Z = \{\mathbf{a}\colon a_2 = 0,\ -1.6 \le a_1 \le 1.6\}\ . \tag{20.23}$$

We know from Eq. (20.17) that the derivative of $V(\mathbf{a})$ is only zero when the velocity is zero, which corresponds to the a_1 axis. Therefore Z consists of the segment of the a_1 axis that falls within G. The set Z is displayed in Figure 20.10.

The set L is the largest invariant set in Z. To find L we need to answer the question: If we start the pendulum from an initial position between -1.6 and 1.6 radians, with zero initial velocity, will the velocity of the pendulum remain zero? Clearly the only such initial condition would be 0 radians (straight down). If we start the pendulum from any other position in Z, the pendulum will start to fall, so the velocity will not remain zero and the trajectory will move out of Z. Therefore, the set L consists only of the origin:

$$L = \{\mathbf{a}\colon \mathbf{a} = 0\}\ . \tag{20.24}$$

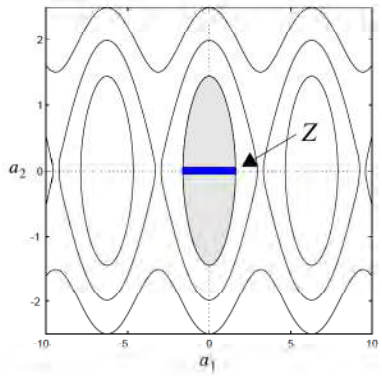


Figure 20.10 Illustration of the Set Z

The set L° is the closure of the intersection of L and G, which in this case is simply L:

$$L^\circ = closure(L \cap G) = L = \{\mathbf{a}: \mathbf{a} = 0\}. \tag{20.25}$$

Therefore, based on LaSalle's corollary, L° is an attractor (asymptotically stable point) and G is in its region of attraction. This means that any trajectory that starts in G will decay to the origin.

Now suppose that we had taken a bigger region for Ω_η, such as

$$\Omega_{300} = \{\mathbf{a}: (V(\mathbf{a}) < 300)\}. \tag{20.26}$$

This set is shown in gray in Figure 20.11.

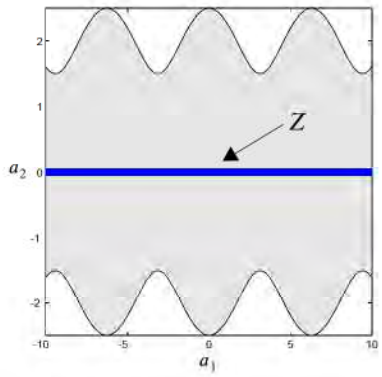


Figure 20.11 Illustration of $G = \Omega_{300}$ (Gray) and Z

We let $G = \Omega_{300}$, since Ω_{300} has only one component. The set Z is given by

$$Z = \{\mathbf{a}: a_2 = 0\}, \tag{20.27}$$

which is shown by the blue bar on the horizontal axis of Figure 20.11. Thus, it follows that

$$L^\circ = L = \{\mathbf{a}: a_1 = \pm n\pi, a_2 = 0\}. \tag{20.28}$$

This is because there are now several different positions within the set Z where we can place the pendulum, without causing the velocity to become nonzero. The pendulum can be pointing directly up or directly down. This corresponds to the positions $\pm n\pi$ for any integer n. If we place the pendulum in any of these positions, with zero velocity, then the pendulum will remain stationary. We can show this by setting the derivatives equal to zero in Eq. (20.14) and Eq. (20.15).

$$\frac{da_1}{dt} = a_2 = 0, \tag{20.29}$$

$$\left(\frac{da_2}{dt} = -\sin(a_1) - 0.2a_2 = -\sin(a_1) = 0\right) \Rightarrow (a_1 = \pm n\pi) \tag{20.30}$$

For this choice of $G = \Omega_{300}$ we can say very little about where the trajectory will converge. We tried to increase the size of our known region of attraction for the origin, but this G is a region of attraction for all of the equilibrium points. We made G too large. The set L° is illustrated by the blue dots in Figure 20.12.

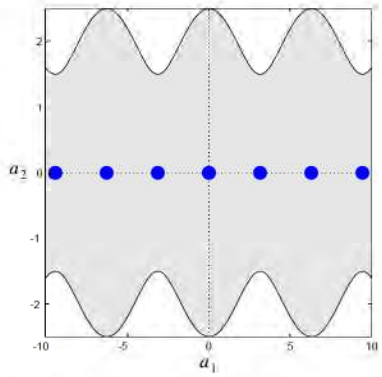


Figure 20.12 The Set L°

We cannot tell which of the equilibrium points (blue dots) will attract the trajectory. All we can say is that if we start somewhere in Ω_{300}, one of the equilibrium points will attract the system solution, but we cannot say for sure which one it will be. Consider, for instance, the trajectory shown in

Figure 20.13. This shows the pendulum response for an initial position of 2 radians and an initial velocity of 1.5 radians per second. This time the pendulum had enough velocity to go over the top, and it converged to the equilibrium point at 2π radians.



Now that we have discussed LaSalle's Invariance Theorem, you might want to experiment some more with the pendulum, in order to investigate the regions of attraction for the various stable points. To experiment with the pendulum, use the Neural Network Design Demonstration Dynamic System (**nnd17ds**).

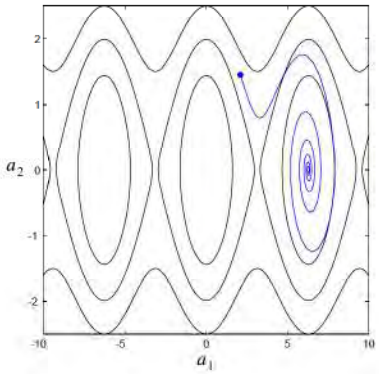


Figure 20.13 Pendulum Trajectory for Different Starting Conditions

Comments

The keys to LaSalle's theorem are the choices of the Lyapunov function V and the set G. We want G to be as large as possible, because that will indicate the region of attraction. However, we want to choose V so that the set Z, which will contain the attractor set, is as small as possible.

For instance, we could try $V = 0$. This is a Lyapunov function for the entire space $\Re^n$, since its derivative is zero (and therefore doesn't change sign) everywhere. However, it gives us no information since $Z = \Re^n$.

Notice that if V_1 and V_2 are both Lyapunov functions on G, and dV_1/dt and dV_2/dt have the same sign, then $V = V_1 + V_2$ is also a Lyapunov function, where $Z = Z_1 \cap Z_2$. If Z is smaller than both Z_1 and Z_2, then Z is a "better" Lyapunov function than either V_1 or V_2. V is always at least as good as either V_1 or V_2, since Z can never be larger than the smaller of Z_1 and Z_2. Therefore, if you have found two different Lyapunov functions and their derivatives have the same sign, then add them together and you may have a better function. The best Lyapunov function for a given system is the one that has the smallest attractor set and the largest region of attraction.

Summary of Results

Stability Concepts

Definitions

Definition 1: *Stability* (in the sense of Lyapunov)

The origin is a stable equilibrium point if for any given value $\varepsilon > 0$ there exists a number $\delta(\varepsilon) > 0$ such that if $\|\mathbf{a}(0)\| < \delta$, then the resulting motion $\mathbf{a}(t)$ satisfies $\|\mathbf{a}(t)\| < \varepsilon$ for $t > 0$.

Definition 2: *Asymptotic Stability*

The origin is an asymptotically stable equilibrium point if there exists a number $\delta > 0$ such that whenever $\|\mathbf{a}(0)\| < \delta$ the resulting motion satisfies $\|\mathbf{a}(0)\| \to 0$ as $t \to \infty$.

Definition 3: *Positive Definite*

A scalar function $V(\mathbf{a})$ is positive definite if $V(\mathbf{0}) = 0$ and $V(\mathbf{a}) > 0$ for $\mathbf{a} \neq \mathbf{0}$.

Definition 4: *Positive Semidefinite*

A scalar function $V(\mathbf{a})$ is positive semidefinite if $V(\mathbf{a}) \geq 0$ for all $\mathbf{a}$.

Lyapunov Stability Theorem

Consider the autonomous (unforced, no explicit time dependence) system

$$\frac{d\mathbf{a}}{dt} = \mathbf{g}(\mathbf{a}).$$

The Lyapunov stability theorem can then be stated as follows.

Theorem 1: *Lyapunov Stability Theorem*

If a positive definite function $V(\mathbf{a})$ can be found such that $dV(\mathbf{a})/dt$ is negative semidefinite, then the origin ($\mathbf{a} = \mathbf{0}$) is stable for this system. If a positive definite function $V(\mathbf{a})$ can be found such that $dV(\mathbf{a})/dt$ is negative definite, then the origin ($\mathbf{a} = \mathbf{0}$) is asymptotically stable. In each case, V is called a Lyapunov function of the system.

LaSalle's Invariance Theorem

Definitions

Definition 5: *Lyapunov Function*

Let V be a continuously differentiable function from $\Re^n$ to $\Re$. If G is any subset of $\Re^n$, we say that V is a Lyapunov function on G for the system $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$ if

$$\frac{dV(\mathbf{a})}{dt} = (\nabla V(\mathbf{a}))^T \mathbf{g}(\mathbf{a})$$

does not change sign on G.

Definition 6: *Set Z*

$$Z = \{\mathbf{a}: dV(\mathbf{a})/dt = 0, \ \mathbf{a} \text{ in the closure of } G\}. \tag{20.31}$$

Definition 7: *Invariant Set*

A set of points G in $\Re^n$ is *invariant* with respect to $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$ if every solution of $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$ starting in G remains in G for all time.

Definition 8: *Set L*

L is defined as the largest invariant set in Z.

Theorem

Theorem 2: *LaSalle's Invariance Theorem*

If V is a Lyapunov function on G for $d\mathbf{a}/dt = \mathbf{g}(\mathbf{a})$, then each solution $\mathbf{a}(t)$ that remains in G for all $t > 0$ approaches $L^\circ = L \cup \{\infty\}$ as $t \to \infty$. (G is a basin of attraction for L, which has all of the stable points.) If all trajectories are bounded, then $\mathbf{a}(t) \to L$ as $t \to \infty$.

Corollary 1: *LaSalle's Corollary*

Let G be a component (one connected subset) of

$$\Omega_\eta = \{\mathbf{a}: V(\mathbf{a}) < \eta\}. \tag{20.32}$$

Assume that G is bounded, $dV(\mathbf{a})/dt \le 0$ on the set G, and let the set $L^\circ = closure(L \cap G)$ be a subset of G. Then L° is an attractor, and G is in its region of attraction.

Solved Problems

P20.1 Test the stability of the origin for the following system.

$$da_1/dt = -a_1 + (a_2)^2$$

$$da_2/dt = -a_2(a_1 + 1)$$

The basic job here is to find a Lyapunov $V(\mathbf{a})$ that is positive definite and has a derivative that is negative semidefinite or, better yet, negative definite. (The latter is a stronger condition.)

Let us try $V(\mathbf{a}) = (a_1)^2 + (a_2)^2$. The derivative of $V(\mathbf{a})$ is

$$\frac{dV(\mathbf{a})}{dt} = (\nabla V)^T\left(\frac{d\mathbf{a}}{dt}\right) = \frac{\partial V}{\partial a_1}\left(\frac{da_1}{dt}\right) + \frac{\partial V}{\partial a_2}\left(\frac{da_2}{dt}\right),$$

or

$$\frac{dV(\mathbf{a})}{dt} = 2a_1(-a_1 + (a_2)^2) + 2a_2(-a_2(a_1 + 1)) = -2(a_1)^2 - 2(a_2)^2.$$

The derivative $dV(\mathbf{a})/dt$ is negative definite. Therefore, the origin is asymptotically stable.

P20.2 Test the stability of the origin for the following system.

$$da_1/dt = -(a_1)^5$$

$$da_2/dt = -5(a_2)^7$$

Let us try $V(\mathbf{a}) = (a_1)^2 + (a_2)^2$. Then we have

$$\frac{dV(\mathbf{a})}{dt} = 2a_1(-(a_1)^5) + 2a_2(-5(a_2)^7) = -2(a_1)^6 - 10(a_2)^8.$$

Here again, $dV(\mathbf{a})/dt$ is negative definite, and therefore the origin is asymptotically stable.

P20.3 Consider the mechanical system shown in Figure P20.1. This is a spring-mass-damper system, with a nonlinear spring. We will define $a_1 = x$ and $a_2 = dx/dt$. Then the equations of motion are

$$da_1/dt = a_2,$$

$$da_2/dt = -(a_1)^3 - a_2 \quad \text{(nonlinear spring)}.$$

Consider the candidate Lyapunov function

$$V(\mathbf{a}) = \frac{1}{4}(a_1)^4 + \frac{1}{2}(a_2)^2.$$

Use the corollary to LaSalle's invariance theorem to provide as much information as possible about the equilibrium points and basins of attraction.

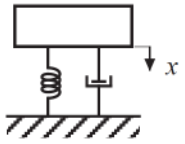


Figure P20.1 Mechanical System

First calculate the derivative of $V(\mathbf{a})$ as

$$\frac{dV(\mathbf{a})}{dt} = \frac{\partial V}{\partial a_1}\left(\frac{da_1}{dt}\right) + \frac{\partial V}{\partial a_2}\left(\frac{da_2}{dt}\right) = (a_1)^3 a_2 + a_2(-(a_1)^3 - a_2) = -(a_2)^2.$$

Thus, dV/dt does not change sign on $\Re^2$.

Now let us define

$$G = \Omega_\eta = \{\mathbf{a}: V(\mathbf{a}) \le \eta\}$$

and consider the case for $\eta = 1$. A contour plot of $V(\mathbf{a})$ is shown in Figure P20.2. The set Ω_1 is indicated in blue on the plot.

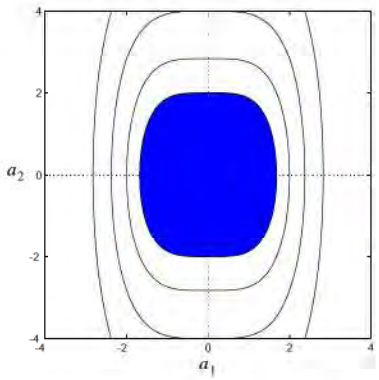


Figure P20.2 Contour Plot of $V(\mathbf{a})$ and Ω_1

Now we need to determine the set Z.

$$Z = \{\mathbf{a}: dV/dt = 0, \mathbf{a} \text{ in the closure of } G\} = \{\mathbf{a}: a_2 = 0, \mathbf{a} \text{ in the closure of } G\}$$

or

$$Z = \{\mathbf{a}: a_2 = 0, \ -\sqrt{2} \le a_1 \le \sqrt{2}\}$$

Next we find the set L. Since $\mathbf{a} = 0$ is the only invariant set,

$$L = \{\mathbf{a}: \ a_1 = 0, \ a_2 = 0\}.$$

Therefore, the origin,

$$\mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

is an attractor and Ω_1 is in its region of attraction.

Further, we can increase η to show that the entire $\Re^2$ is the basin of attraction for the origin.

Figure P20.3 shows the response of the spring-mass-damper from an initial position of 2 and an initial velocity of 2. Note that the trajectory is parallel to the contour lines when the trajectory crosses the a_2 axis. This agrees with our earlier result, which showed that the derivative of the Lyapunov function was zero whenever $a_2 = 0$. Fortunately, the a_2 axis is not an invariant set (except for the origin); therefore the trajectory is only attracted to the origin.

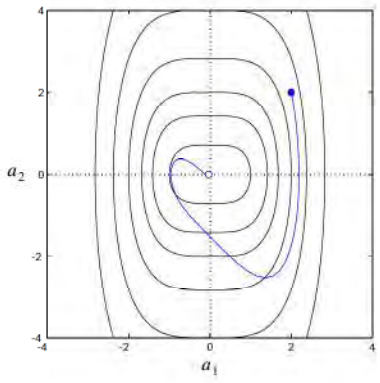


Figure P20.3 Spring-Mass-Damper Response

P20.4 Consider the following nonlinear system:

$$da_1/dt = a_1((a_1)^2 + (a_2)^2 - 4) - a_2$$

$$da_2/dt = a_1 + a_2((a_1)^2 + (a_2)^2 - 4).$$

This system has two invariant sets, the origin

$$\{\mathbf{a}:\ \mathbf{a} = \mathbf{0}\ \},$$

and the circle

$$\{\mathbf{a}:\ (a_1)^2 + (a_2)^2 = 4\ \}.$$

Assuming the candidate Lyapunov function

$$V(\mathbf{a}) = (a_1)^2 + (a_2)^2,$$

use LaSalle's Invariance Theorem to find out as much as you can about the region of attraction for the origin.

Our job, then, is to determine whether or not the given invariant sets represent a stable point or a stable trajectory. Let's first take a look at dV/dt. We recall that

$$\frac{dV(\mathbf{a})}{dt} = \frac{\partial V}{\partial a_1}\left(\frac{da_1}{dt}\right) + \frac{\partial V}{\partial a_2}\left(\frac{da_2}{dt}\right),$$

and substitute for the various terms to give

$$\frac{dV(\mathbf{a})}{dt} = 2a_1[a_1((a_1)^2 + (a_2)^2 - 4) - a_2] + 2a_2[a_1 + a_2((a_1)^2 + (a_2)^2 - 4)].$$

This can be simplified to

$$\frac{dV(\mathbf{a})}{dt} = 2((a_1)^2 + (a_2)^2)((a_1)^2 + (a_2)^2 - 4).$$

Thus, dV/dt is zero at $\mathbf{a} = \mathbf{0}$ and on the circle $(a_1)^2 + (a_2)^2 = 4$.

We now pick G, a region of attraction. Is there a change of sign of dV/dt over all $\Re^2$? Yes, there is. As we go from outside the circle of radius 2 to its interior, the sign of dV/dt changes from positive to negative. So dV/dt is negative semidefinite inside the circle $(a_1)^2 + (a_2)^2 = 4$. Let's pick a G inside this circle, so that the circle will not be included. The following set will do.

$$G = \Omega_1 = \{\mathbf{a}: V(\mathbf{a}) \leq 1\}$$

Now we consider Ω_1. There are just two places that $dV/dt = 0$, and the only point inside Ω_1 is $\mathbf{a} = \mathbf{0}$. Therefore,

$$Z = \{\mathbf{a}: \quad a_1 = 0, a_2 = 0\}$$

and

$$L^\circ = L = Z.$$

The origin is the attractor, and Ω_1 is in its region of attraction.We can use the same arguments to show that the region of attraction for the origin includes all points inside the circle $(a_1)^2 + (a_2)^2 = 4$.

Figure P20.4 displays two trajectories for this system, one that begins inside the circle $(a_1)^2 + (a_2)^2 = 4$, and one that begins outside the circle. Although the circle is an invariant set, it is not an attractor. The only attractor for this system is the origin.

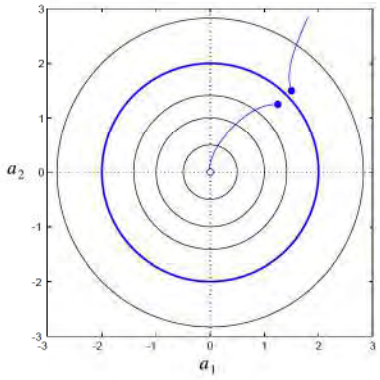


Figure P20.4 Sample Trajectories for Problem P20.4

P20.5 Consider the following nonlinear system.

$$da(t)/(dt) = -(a(t) - 1)(a(t) - 2)$$

i. Find any equilibrium points for this system.

ii. Use the following candidate Lyapunov function to obtain whatever information you can about the regions of attraction for the equilibrium points found in part (i). (Hint: Use the corollary to LaSalle's Invariance Theorem.)

$$V(\mathbf{a}) = (a-2)^2$$

i. To find the equilibrium points, we set $da(t)/dt = 0$.

$$0 = -(a-1)(a-2) \quad \Rightarrow \quad a = 1, a = 2 \quad \text{are equilibrium points}$$

ii. To use LaSalle's corollary, we need to find dV/dt.

$$\frac{dV}{dt} = \frac{\partial V}{\partial a}\left(\frac{da}{dt}\right) = 2(a-2)[-(a-1)(a-2)] = -2(a-1)(a-2)^2$$

Now we let

$$G = \Omega_\eta = \{a: V(a) < \eta\}.$$

For example, try $\eta = 0.5$. This gives

$$G = \Omega_{0.5} = \{a: (a-2)^2 < 0.5\}.$$

Note that a solution of $(a-2)^2 < 0.5$ yields

$$\pm(a-2) < \sqrt{0.5} \text{ or } 1.3 < a < 2.7.$$

Thus, dV/dt is negative definite on G.

Next we need to find the set Z, which contains those points within G where dV/dt is zero. There are two points where dV/dt is zero, $a = 1$ and $a = 2$. Only one of these falls within G. Therefore

$$Z = \{a: \ a = 2\}.$$

Now we need to find L, the largest invariant set in Z. There is only one point in Z, and it is an equilibrium point. Thus

$$L^\circ = L = Z.$$

This means that G is in the region of attraction for 2.

We can use the same arguments with values of η up to 1.0. So we can say that the region for attraction for $a = 2$ must include at least

$$\{a: \ 1 < a < 3\}.$$

What if we consider those regions where $\eta > 1$? Then Z includes both 1 and 2, and dV/dt will change sign on G. Therefore we cannot say anything about the region of attraction for $a = 1$, using this Lyapunov function and the corollary to LaSalle's Invariance Theorem.

Figure P20.5 displays some typical responses for this system. Here we can see that the equilibrium point $a = 1$ is actually unstable. Any initial condition above $a = 1$ converges to $a = 2$. Anything less than $a = 1$ goes to minus infinity.

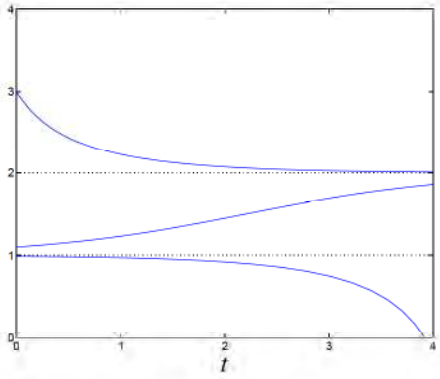


Figure P20.5 Stable and Unstable Responses for Problem P20.5

Epilogue

In this chapter we have presented the concept of stability, as applied to dynamic systems. For nonlinear dynamic systems, like recurrent neural networks, we do not talk about the stability of the system. Rather, we discuss the stability of certain system trajectories and, in particular, equilibrium points.

There were two main stability theorems discussed in this chapter. The first is the Lyapunov Stability Theorem, which introduces the concept of generalized energy — the Lyapunov function. The concept behind this theorem is that if a system's "energy" is always decreasing, then it will eventually stabilize at a point of minimum "energy."

The second theorem presented was LaSalle's Invariance Theorem, which is an enhancement of the Lyapunov Stability Theorem. There are two key improvements made by LaSalle. The first is a clarification of the cases in which the Lyapunov function does not decrease throughout the state space, but stays constant in some regions. LaSalle's theorem introduced the concept of an invariant set to identify those regions that can trap the system trajectory. The second improvement made by LaSalle's theorem is that, in addition to indicating the stability of equilibrium points, it also gave information about the regions of attraction of each stable point.

The ideas presented in this chapter are important tools for the analysis of recurrent neural networks, like the Grossberg networks of Chapters 18 and 19. (See [CoGr83] for an application of LaSalle's Invariance Theorem to recurrent neural networks.) In Chapter 21 we will use LaSalle's theorem to explain the operation of the Hopfield network.

Further Reading

[Brog91] W. L. Brogan, *Modern Control Theory,* 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1991.

This is a well-written book on the subject of linear systems. The first half of the book is devoted to linear algebra. It also has good sections on the solution of linear differential equations and the stability of linear and nonlinear systems. It has many worked problems.

[CoGr83] M. A. Cohen and S. Grossberg, "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 13, no. 5, pp. 815–826, 1983.

Cohen and Grossberg apply LaSalle's Invariance Theorem to the analysis of the stability of competitive neural networks. The network description is very general, and the authors show how their analysis can be applied to many different types of recurrent neural networks.

[Lasa67] J. P. LaSalle, "An invariance principle in the theory of stability," in *Differential Equations and Dynamic Systems*, J. K. Hale and J. P. LaSalle, eds., New York: Academic Press, pp. 277–286, 1967.

This article provides a unified presentation of Lyapunov's stability theory, including several extensions. It introduces LaSalle's Invariance Theorem and various corollaries.

[SlLi91] J.-J. E. Slotine and W. Li, *Applied Nonlinear Control*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

This text is an introduction to nonlinear control systems. A significant portion of the book is devoted to the analysis of nonlinear dynamic systems. A number of stability theorems are presented and demonstrated.

Exercises

E20.1 Use Lyapunov's Stability Theorem to test the stability of the origin for the following systems.

i.
$$da_1/dt = -(a_1)^3 + a_2$$
$$da_2/dt = -a_1 - a_2$$

ii.
$$da_1/dt = -a_1 + (a_2)^2$$
$$da_2/dt = -a_2(a_1 + 1)$$

E20.2 Consider the following nonlinear system:

$$da_1/dt = a_2 - 2a_1((a_1)^2 + (a_2)^2),$$
$$da_2/dt = -a_1 - 2a_2((a_1)^2 + (a_2)^2).$$

i. Use Lyapunov's Stability Theorem and the candidate Lyapunov function shown below to investigate the stability of the origin.

$$V(\mathbf{a}) = \alpha(a_1)^2 + \beta(a_2)^2$$



ii. Check your stability result from part (i) by writing a MATLAB M-file to simulate the response of this system for several different initial conditions. Use the **ode45** routine. Plot the responses.

E20.3 Consider the following nonlinear system:

$$da/dt = a(a + 1),$$

i. Find any equilibrium points.

ii. The following Lyapunov function is proposed. Show that this is a valid Lyapunov function for use in Lasalle's invariance theorem.

$$V(a) = -(2a^3 + 3a^2).$$

iii. Use the corollary to Lasalle's theorem and the proposed Lyapunov function to provide as much information as you can about the stable equilibrium points and their basins of attraction. Identify the sets Z, G and L. Use graphs wherever possible.

E20.4 Repeat E20.3 for the following systems and Lyapunov functions. (In some cases, it may be useful to sketch the Lyapunov functions.)

i. $da/dt = (a-2)(a+1)$, $V(a) = (a+1)^2$

ii. $da/dt = a(a+1)$, $V(a) = -(2a^3 + 3a^2)$

iii. $da/dt = a(a+2)$, $V(a) = -a^3/3 - a^2$

iv. $da/dt = -a(a-1)$, $V(a) = 2a^3 - 3a^2$

v. $da/dt = \cos(a)$, find a $V(a)$

vi. $da/dt = \sin(a)$, find a $V(a)$

E20.5 Consider the following nonlinear system:

$$da_1/dt = a_2,$$

$$da_2/dt = -a_2(1-a_2)^2 - a_1.$$

We want to use the corollary to Lasalle's invariance theorem to locate attractors and find out as much as we can about the basins of attraction, using the following Lyapunov function.

$$V(\mathbf{a}) = (a_1)^2 + (a_2)^2.$$

i. Find any equilibrium points.

ii. Find $dV(\mathbf{a})/dt$.

iii. Choose a set G.

iv. Find the corresponding set Z.

v. Find the set L.

vi. What have you learned about the attractors of this system and the basins of attraction? Can you learn more by modifying the set G? Explain.



vii. Check your results by writing a MATLAB M-file to simulate the response of this system for several different initial conditions. Use the `ode45` routine. Plot the responses.

E20.6 Consider the following nonlinear system:

$$da_1/dt = a_2,$$

$$da_2/dt = -a_1 - (a_2)^3 .$$

i. Find any equilibrium points.

ii. Find as much information about the stability of the equilibrium points as possible, using the corollary to LaSalle's theorem and the candidate Lyapunov function

$$V(\mathbf{a}) = (a_1)^2 + (a_2)^2 .$$



iii. Check your results from parts (i) and (ii) by writing a MATLAB M-file to simulate the response of this system for several different initial conditions. Use the **ode45** routine. Plot the responses.

E20.7 Consider the following nonlinear system:

$$da/dt = (1-a)(1+a) = 1 - a^2 .$$

i. Find any equilibrium points.

ii. Find a suitable Lyapunov function. (Hint: Start with a form for dV/dt and work backward to find V.)

iii. Sketch the Lyapunov function.

iv. Use the corollary to LaSalle's theorem and the Lyapunov function of part (ii) to find as much information as possible about regions of attraction. Use graphs wherever possible.

(Hint: The graph shown in Figure E20.1 may be helpful.)

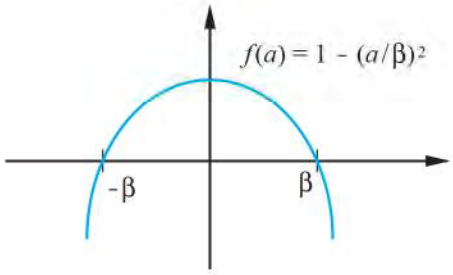


Figure E20.1 Helpful Function for Exercise E20.7

E20.8 Consider the following nonlinear system:

$$da_1/dt = a_2 - a_1((a_1)^4 + 2(a_2)^2 - 10),$$

$$da_2/dt = -(a_1)^3 - 3(a_2)^5((a_1)^4 + 2(a_2)^2 - 10).$$



i. Find any invariant sets. (You may want to simulate this system using MATLAB in order to help identify the invariant sets.)

ii. Using the candidate Lyapunov function shown below and the corollary to LaSalle's theorem, investigate the stability of the invariant sets you found in part (i).

$$V(\mathbf{a}) = ((a_1)^4 + 2(a_2)^2 - 10)^2$$

E20.9 Consider the following system:

$$\frac{d\mathbf{a}}{dt} = \begin{bmatrix} -1 & 0 \\ 0 & -2 \end{bmatrix}\mathbf{a} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

i. Find any equilibrium points.

ii. Find a Lyapunov function and identify attractors and basins of attraction. Use the corollary to Lasalle's theorem and carefully identify and graph the sets Ω_η, G, Z and L.

E20.10 For the nonlinear system

$$da_1/dt = a_2 + a_1(1 - (a_1)^2 - (a_2)^2),$$

$$da_2/dt = -a_1 + a_2(1 - (a_1)^2 - (a_2)^2),$$

we know that the following sets are invariant:

$$\{\mathbf{a} | \mathbf{a} = 0\},$$

$$\{\mathbf{a} | (a_1)^2 + (a_2)^2 = 1\}.$$

The following Lyapunov function is proposed:

$$V(\mathbf{a}) = ((a_1)^2 + (a_2)^2 - 1)^2.$$

Use the corollary to Lasalle's theorem to find out as much as possible about the basins of attraction for the two invariant sets given above, and graph the sets Ω_η, G, Z and L.

E20.11 Consider the system

$$da_1/dt = -\cos(a_2) - a_1,$$

$$da_2/dt = a_1.$$

i. Find any equilibrium points.

ii. The following Lyapunov function is proposed. Show that this is a valid Lyapunov function for use in Lasalle's invariance theorem.

$$V(\mathbf{a}) = \frac{(a_1)^2}{2} + \sin(a_2)$$

iii. Use Lasalle's theorem to find out as much information as you can about the stable equilibrium points and their basins of attraction. (Make a rough sketch of the contour plot for $V(\mathbf{a})$ to assist you.)

21 Hopfield Network

Objectives	21-1
Theory and Examples	21-2
Hopfield Model	21-3
Lyapunov Function	21-5
Invariant Sets	21-7
Example	21-8
Hopfield Attractors	21-11
Effect of Gain	21-13
Hopfield Design	21-16
Content-Addressable Memory	21-16
Hebb Rule	21-19
Lyapunov Surface	21-23
Summary of Results	21-25
Solved Problems	21-27
Epilogue	21-37
Further Reading	21-38
Exercises	21-41

Objectives

This chapter will discuss the Hopfield recurrent neural network — a network that was highly influential in bringing about the resurgence of neural network research in the early 1980s. We will begin with a description of the network, and then we will show how Lyapunov stability theory can be used to analyze the network operation. Finally, we will demonstrate how the network can be designed to behave as an associative memory.

This chapter brings together many topics discussed in previous chapters: the discrete-time Hopfield network (Chapter 3), eigenvalues and eigenvectors (Chapter 6); associative memory and the Hebb rule (Chapter 7); Hessian matrices, conditions for optimality, quadratic functions and surface and contour plots (Chapter 8); steepest descent and phase plane trajectories (Chapter 9); continuous-time recurrent networks (Chapter 18); and Lyapunov's Stability Theorem and LaSalle's Invariance Theorem (Chapter 20). This chapter is, in some ways, a culmination of all our previous efforts.

Theory and Examples

Much of the resurgence of interest in neural networks during the early 1980s can be attributed to the work of John Hopfield. As a well-known Cal. Tech. physicist, Hopfield's visibility and scientific credentials lent renewed credibility to the neural network field, which had been tarnished by the hype of the mid-1960s. Early in his career he studied the interaction between light and solids. Later he focused on the mechanism of electron transfer between biological molecules. One can imagine that his academic study in physics and mathematics, combined with his later experiences in biology, prepared him uniquely for the conception and presentation of his neural network contribution.

Hopfield wrote two highly influential papers in 1982 [Hopf82] and 1984 [Hopf84]. Many of the ideas in these papers were based on the previous work of other researchers, such as the neuron model of McCulloch and Pitts [McPi43], the additive model of Grossberg [Gros67], the linear associator of Anderson [Ande72] and Kohonen [Koho72] and the Brain-State-in-a-Box network of Anderson, Silverstein, Ritz and Jones [AnSi77]. However, Hopfield's papers are very readable, and they bring together a number of important ideas and present them with a clear mathematical analysis (including the application of Lyapunov stability theory).

There are several other reasons why Hopfield's papers have had such an impact. First, he identified a close analogy between his neural network and the Ising model of magnetic materials, which is used in statistical physics. This brought a significant amount of existing theory to bear on the analysis of neural networks, and it encouraged many physicists, as well as other scientists and engineers, to turn their attention to neural network research.

Hopfield also had close contacts with VLSI chip designers, because of his long association with AT&T Bell Laboratories. As early as 1987, Bell Labs had successfully developed neural network chips based on the Hopfield network. One of the main promises of neural networks is their suitability for parallel implementation in VLSI and optical devices. The fact that Hopfield addressed the implementation issues of his networks distinguished him from most previous neural network researchers.

Hopfield emphasized practicality, both in the implementation of his networks and in the types of problems they solved. Some of the applications that he described in his early papers include content-addressable memory (which we will discuss later in this chapter), analog-to-digital conversion [TaHo86], and optimization [HoTa85] (as in the traveling salesman problem).

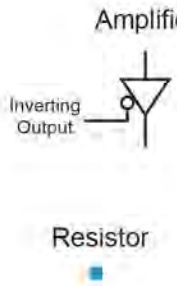
In the next section we will present the Hopfield model. We will use the continuous-time model from the 1984 paper [Hopf84]. Then we will apply Lyapunov stability theory and LaSalle's Invariance Theorem to the analy-

sis of the Hopfield model. In the final section we will demonstrate how the Hebb rule can be used to design Hopfield networks as content-addressable memories.

Hopfield Model

Hopfield Model

In keeping with his practical viewpoint, Hopfield presented his model as an electrical circuit. The basic *Hopfield model* (see [Hopf84]) is shown in Figure 21.1.



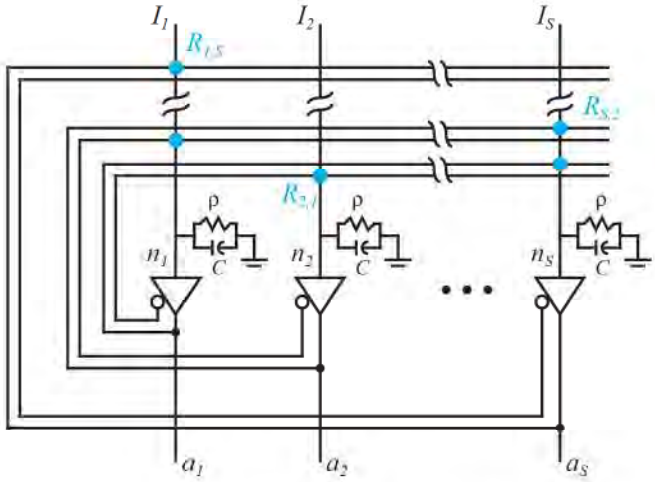


Figure 21.1 Hopfield Model

Each neuron is represented by an operational amplifier and its associated resistor/capacitor network. There are two sets of inputs to the neurons. The first set, represented by the currents $I_1, I_2, \ldots$, are constant external inputs. The other set consists of feedback connections from other op-amps. For instance, the second output, a_2, is fed to resistor $R_{S,2}$, which is connected, in turn, to the input of amplifier S. Resistors are, of course, only positive, but a negative input to a neuron can be obtained by selecting the inverted output of a particular amplifier. (In Figure 21.1, the inverting output of the first amplifier is connected to the input of the second amplifier through resistor $R_{2,1}$.)

The equation of operation for the Hopfield model, derived using Kirchhoff's current law, is

$$C\frac{dn_i(t)}{dt} = \sum_{j=1}^{S} T_{i,j}a_j(t) - \frac{n_i(t)}{R_i} + I_i, \qquad (21.1)$$

where n_i is the input voltage to the ith amplifier, a_i is the output voltage of the ith amplifier, C is the amplifier input capacitance and I_i is a fixed input current to the ith amplifier. Also,

$$|T_{i,j}| = \frac{1}{R_{i,j}}, \frac{1}{R_i} = \frac{1}{\rho} + \sum_{j=1}^{S} \frac{1}{R_{i,j}}, \; n_i = f^{-1}(a_i) \text{ (or } a_i = f(n_i)\text{)}, \tag{21.2}$$

where $f(n)$ is the amplifier characteristic. Here and in what follows we will assume that the circuit is symmetric, so that $T_{i,j} = T_{j,i}$.

The amplifier transfer function, $a_i = f(n_i)$, is ordinarily a sigmoid function. Both this sigmoid function and its inverse are assumed to be increasing functions. We will provide a specific example of a suitable transfer function later in this chapter.

If we multiply both sides of Eq. (21.1) by R_i, we obtain

$$R_i C \frac{dn_i(t)}{dt} = \sum_{j=1}^{S} R_i T_{i,j} a_j(t) - n_i(t) + R_i I_i. \tag{21.3}$$

This can be transformed into our standard neural network notation if we define

$$\varepsilon = R_i C, \; w_{i,j} = R_i T_{i,j} \text{ and } b_i = R_i I_i. \tag{21.4}$$

Now Eq. (21.3) can be rewritten as

$$\varepsilon \frac{dn_i(t)}{dt} = -n_i(t) + \sum_{j=1}^{S} w_{i,j} a_j(t) + b_i. \tag{21.5}$$

In vector form we have

$$\varepsilon \frac{d\mathbf{n}(t)}{dt} = -\mathbf{n}(t) + \mathbf{W}\mathbf{a}(t) + \mathbf{b}. \tag{21.6}$$

and

$$\mathbf{a}(t) = \mathbf{f}(\mathbf{n}(t)). \tag{21.7}$$

The resulting Hopfield network is displayed in Figure 21.2.

Thus, Hopfield's original network of S operational amplifier circuits can be represented conveniently in our standard network notation. Note that the input vector $\mathbf{p}$ determines the initial network output. This form of the Hopfield network is used for associative memory networks, as will be discussed at the end of this chapter.

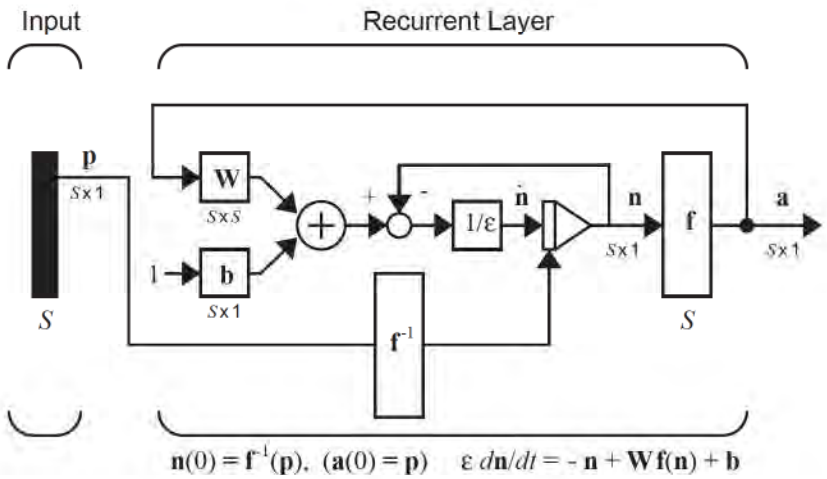


Figure 21.2 Hopfield Network

Lyapunov Function

The application of Lyapunov stability theory to the analysis of recurrent networks was one of the key contributions of Hopfield. (Cohen and Grossberg also used Lyapunov theory for the analysis of competitive networks at about the same time [CoGr83].) In this section we will demonstrate how LaSalle's Invariance Theorem, which was presented in Chapter 20, can be used with the Hopfield network. The first step in using LaSalle's theorem is to choose a Lyapunov function. Hopfield suggested the following function:

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} + \sum_{i=1}^{S}\left\{\int_0^{a_i} f^{-1}(u)\,du\right\} - \mathbf{b}^T\mathbf{a}. \tag{21.8}$$

Hopfield's choice of this particular Lyapunov candidate is one of his key contributions. Notice that the first and third terms make up a quadratic function. In a later section of this chapter we will use our previous results on quadratic functions to help develop some insight into this Lyapunov function.

To use LaSalle's theorem, we will need to evaluate the derivative of $V(\mathbf{a})$. For clarity, we will consider each of the three terms of $V(\mathbf{a})$ separately. Using Eq. (8.37), the derivative of the first term is

$$\frac{d}{dt}\left\{-\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a}\right\} = -\frac{1}{2}\nabla[\mathbf{a}^T\mathbf{W}\mathbf{a}]^T\frac{d\mathbf{a}}{dt} = -[\mathbf{W}\mathbf{a}]^T\frac{d\mathbf{a}}{dt} = -\mathbf{a}^T\mathbf{W}\frac{d\mathbf{a}}{dt}. \quad (21.9)$$

The second term in $V(\mathbf{a})$ consists of a sum of integrals. If we consider one of these integrals, we find

$$\frac{d}{dt}\left\{\int_0^{a_i} f^{-1}(u)du\right\} = \frac{d}{da_i}\left\{\int_0^{a_i} f^{-1}(u)du\right\}\frac{da_i}{dt} = f^{-1}(a_i)\frac{da_i}{dt} = n_i\frac{da_i}{dt}. \quad (21.10)$$

The total derivative of the second term in $V(\mathbf{a})$ is then

$$\frac{d}{dt}\left[\sum_{i=1}^{S}\left\{\int_0^{a_i} f^{-1}(u)du\right\}\right] = \mathbf{n}^T\frac{d\mathbf{a}}{dt}. \quad (21.11)$$

Using Eq. (8.36), we can find the derivative of the third term in $V(\mathbf{a})$.

$$\frac{d}{dt}\{-\mathbf{b}^T\mathbf{a}\} = -\nabla[\mathbf{b}^T\mathbf{a}]^T\frac{d\mathbf{a}}{dt} = -\mathbf{b}^T\frac{d\mathbf{a}}{dt} \quad (21.12)$$

Therefore, the total derivative of $V(\mathbf{a})$ is

$$\frac{d}{dt}V(\mathbf{a}) = -\mathbf{a}^T\mathbf{W}\frac{d\mathbf{a}}{dt} + \mathbf{n}^T\frac{d\mathbf{a}}{dt} - \mathbf{b}^T\frac{d\mathbf{a}}{dt} = [-\mathbf{a}^T\mathbf{W} + \mathbf{n}^T - \mathbf{b}^T]\frac{d\mathbf{a}}{dt}. \quad (21.13)$$

From Eq. (21.6) we know that

$$[-\mathbf{a}^T\mathbf{W} + \mathbf{n}^T - \mathbf{b}^T] = -\varepsilon\left[\frac{d\mathbf{n}(t)}{dt}\right]^T. \quad (21.14)$$

This allows us to rewrite Eq. (21.13) as

$$\frac{d}{dt}V(\mathbf{a}) = -\varepsilon\left[\frac{d\mathbf{n}(t)}{dt}\right]^T\frac{d\mathbf{a}}{dt} = -\varepsilon\sum_{i=1}^{S}\left(\frac{dn_i}{dt}\right)\left(\frac{da_i}{dt}\right). \quad (21.15)$$

Since $n_i = f^{-1}(a_i)$, we can expand the derivative of n_i as follows:

$$\frac{dn_i}{dt} = \frac{d}{dt}[f^{-1}(a_i)] = \frac{d}{da_i}[f^{-1}(a_i)]\frac{da_i}{dt}. \quad (21.16)$$

Now Eq. (21.15) can be rewritten

$$\frac{d}{dt}V(\mathbf{a}) = -\varepsilon \sum_{i=1}^{S} \left(\frac{dn_i}{dt}\right)\left(\frac{da_i}{dt}\right) = -\varepsilon \sum_{i=1}^{S} \left(\frac{d}{da_i}[f^{-1}(a_i)]\right)\left(\frac{da_i}{dt}\right)^2 . \tag{21.17}$$

If we assume that $f^{-1}(a_i)$ is an increasing function, as it would be for an operational amplifier, then

$$\frac{d}{da_i}[f^{-1}(a_i)] > 0 . \tag{21.18}$$

From Eq. (21.17), this implies that

$$\frac{d}{dt}V(\mathbf{a}) \le 0 . \tag{21.19}$$

Thus, if $f^{-1}(a_i)$ is an increasing function, $dV(\mathbf{a})/dt$ is a negative semidefinite function. Therefore, $V(\mathbf{a})$ is a valid Lyapunov function.

Invariant Sets

Now we want to apply LaSalle's Invariance Theorem to determine equilibrium points for the Hopfield network. The first step is to find the set Z (Eq. (20.19)).

$$Z = \{\mathbf{a}: dV(\mathbf{a})/dt = 0, \ \mathbf{a} \text{ in the closure of } G\} \tag{21.20}$$

This set includes all points at which the derivative of the Lyapunov function is zero. For now, let's assume that G is all of $\Re^S$.

We can see from Eq. (21.17) that such derivatives will be zero if the derivatives of all of the neuron outputs are zero.

$$\frac{d\mathbf{a}}{dt} = \mathbf{0} \tag{21.21}$$

However, when the derivatives of the outputs are zero, the circuit is at equilibrium. Thus, those points where the system "energy" is not changing are also points where the circuit is at equilibrium.

This means that the set L, the largest invariant set in Z, is exactly equal to Z.

$$L = Z \tag{21.22}$$

Thus, all points in Z are potential attractors.

Some of these features will be illustrated in the following example.

Example



Consider the following example from Hopfield's original paper [Hopf84]. We will examine a system having an amplifier characteristic

$$a = f(n) = \frac{2}{\pi}\tan^{-1}\left(\frac{\gamma\pi n}{2}\right). \tag{21.23}$$

We can also write this expression as

$$n = \frac{2}{\gamma\pi}\tan\left(\frac{\pi}{2}a\right). \tag{21.24}$$

Assume two amplifiers, with the output of each connected to the input of the other through a unit resistor, so that

$$R_{1,2} = R_{2,1} = 1 \text{ and } T_{1,2} = T_{2,1} = 1. \tag{21.25}$$

Thus we have a weight matrix

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{21.26}$$

If the amplifier input capacitance is also set to 1, we have

$$\varepsilon = R_i C = 1. \tag{21.27}$$

Let us also take $\gamma = 1.4$ and $I_1 = I_2 = 0$. Therefore

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{21.28}$$

Recall from Eq. (21.8) that the Lyapunov function is

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} + \sum_{i=1}^{S}\left\{\int_0^{a_i} f^{-1}(u)du\right\} - \mathbf{b}^T\mathbf{a}. \tag{21.29}$$

The first term of the Lyapunov function, for this example, is

$$-\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} = -\frac{1}{2}\begin{bmatrix} a_1 & a_2 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = -a_1 a_2. \tag{21.30}$$

The third term is zero, because the biases are zero. The ith part of the second term is

$$\int_0^{a_i} f^{-1}(u)du = \frac{2}{\gamma\pi}\int_0^{a_i} \tan\left(\frac{\pi}{2}u\right)du = \frac{2}{\gamma\pi}\left[-\log\left[\cos\left(\frac{\pi}{2}u\right)\right]\frac{2}{\pi}\right]_0^{a_i}. \tag{21.31}$$

This expression can be simplified to

$$\int_0^{a_i} f^{-1}(u)du = -\frac{4}{\gamma\pi^2}\log\left[\cos\left(\frac{\pi}{2}a_i\right)\right]. \tag{21.32}$$

Finally, substituting all three terms into Eq. (21.29), we have our Lyapunov function:

$$V(\mathbf{a}) = -a_1a_2 - \frac{4}{1.4\pi^2}\left[\log\left\{\cos\left(\frac{\pi}{2}a_1\right)\right\} + \log\left\{\cos\left(\frac{\pi}{2}a_2\right)\right\}\right]. \tag{21.33}$$

Now let's write out the network equation (Eq. (21.6)). With $\varepsilon = 1$ and $\mathbf{b} = \mathbf{0}$, it is

$$\frac{d\mathbf{n}}{dt} = -\mathbf{n} + \mathbf{W}\mathbf{f}(\mathbf{n}) = -\mathbf{n} + \mathbf{W}\mathbf{a}. \tag{21.34}$$

If we substitute the weight matrix of Eq. (21.26), this expression can be written as the following pair of equations:

$$dn_1/dt = a_2 - n_1, \tag{21.35}$$

$$dn_2/dt = a_1 - n_2. \tag{21.36}$$

The neuron outputs are

$$a_1 = \frac{2}{\pi}\tan^{-1}\left(\frac{1.4\pi}{2}n_1\right), \tag{21.37}$$

$$a_2 = \frac{2}{\pi}\tan^{-1}\left(\frac{1.4\pi}{2}n_2\right). \tag{21.38}$$

Now that we have found expressions for the system Lyapunov function and the network equation of operation, let's investigate the network behavior. The Lyapunov function contour and a sample trajectory are shown in Figure 21.3.

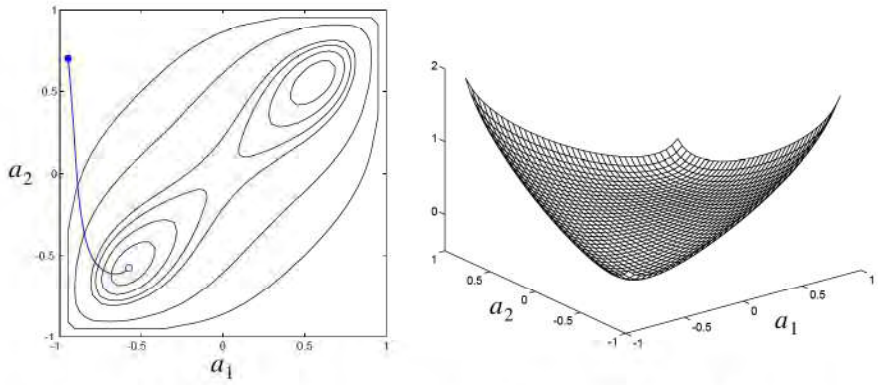


Figure 21.3 Hopfield Example Lyapunov Function and Trajectory

The contour lines in this figure represent constant values of the Lyapunov function. The system has two attractors, one in the lower left and one in the upper right of Figure 21.3. Starting from the upper left, the system converges, as shown by the blue line, to the stable point at the lower left.

Figure 21.4 displays the time response of the two neuron outputs.

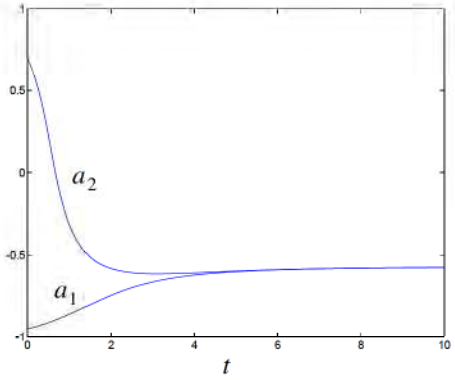


Figure 21.4 Hopfield Example Time Response

Figure 21.5 displays the time response of the Lyapunov function. As expected, it decreases continuously as the equilibrium point is approached.

The system also has an equilibrium point at the origin. If the network is initialized anywhere on a diagonal line drawn from the upper-left corner to the lower-right corner, the solution converges to the origin. Any initial conditions that do not fall on this line, however, will converge to one of the solutions in the lower-left or upper-right corner. The solution at the origin is a saddle point of the Lyapunov function, not a local minimum. We will dis-

cuss this problem in a later section. Figure 21.6 displays a trajectory that converges to the saddle point.

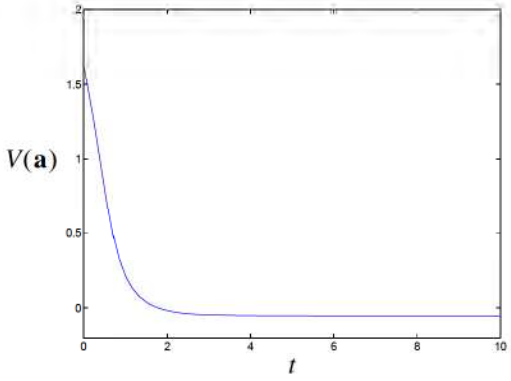


Figure 21.5 Lyapunov Function Response

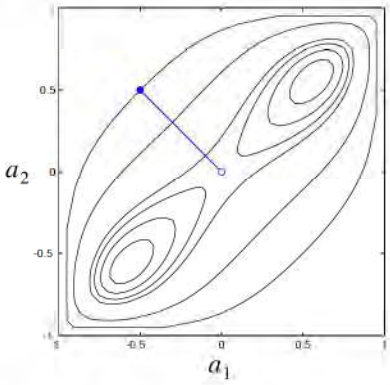


Figure 21.6 Hopfield Convergence to a Saddle Point



To experiment with the Hopfield network, use the Neural Network Design Demonstration Hopfield Network (nnd18hn).

This example has provided some insight into the Hopfield attractors. In the next section we will analyze them more carefully.

Hopfield Attractors

In the example network in the previous section we found that the Hopfield network attractors were stationary points of the Lyapunov function. Now we want to show that this is true in the general case. Recall from Eq. (21.21) that the potential attractors of the Hopfield network satisfy

$$\frac{d\mathbf{a}}{dt} = \mathbf{0}. \tag{21.39}$$

How are these points related to the minima of the Lyapunov function $V(\mathbf{a})$? In Chapter 8 (Eq. (8.27)) we showed that the minima of a function must be stationary points (i.e., gradient equal to zero). The stationary points of $V(\mathbf{a})$ will satisfy

$$\nabla V = \left[\frac{\partial V}{\partial a_1}\ \frac{\partial V}{\partial a_2}\ \cdots\ \frac{\partial V}{\partial a_S}\right]^T = \mathbf{0}, \tag{21.40}$$

where

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} + \sum_{i=1}^{S}\left\{\int_0^{a_i} f^{-1}(u)du\right\} - \mathbf{b}^T\mathbf{a}. \tag{21.41}$$

If we follow steps similar to those we used to derive Eq. (21.13), we can find the following expression for the gradient:

$$\nabla V(\mathbf{a}) = [-\mathbf{W}\mathbf{a} + \mathbf{n} - \mathbf{b}] = -\varepsilon\left[\frac{d\mathbf{n}(t)}{dt}\right]. \tag{21.42}$$

The ith element of the gradient is therefore

$$\frac{\partial}{\partial a_i}V(\mathbf{a}) = -\varepsilon\frac{dn_i}{dt} = -\varepsilon\frac{d}{dt}([f^{-1}(a_i)]) = -\varepsilon\frac{d}{da_i}[f^{-1}(a_i)]\frac{da_i}{dt}. \tag{21.43}$$

Notice, incidentally, that if $f^{-1}(a)$ is linear, Eq. (21.43) implies that

$$\frac{d\mathbf{a}}{dt} = -\alpha\nabla V(\mathbf{a}). \tag{21.44}$$

Therefore, the response of the Hopfield network is steepest descent. Thus, if you are in a region where $f^{-1}(\mathbf{a})$ is approximately linear, the network solution approximates steepest descent.

We have assumed that the transfer function and its inverse are monotonic increasing functions. Therefore,

$$\frac{d}{da_i}[f^{-1}(a_i)] > 0. \tag{21.45}$$

From Eq. (21.43), this implies that those points for which

$$\frac{d\mathbf{a}(t)}{dt} = \mathbf{0}, \tag{21.46}$$

will also be points where

$$\nabla V(\mathbf{a}) = \mathbf{0}. \tag{21.47}$$

Therefore, the attractors, which are members of the set L and satisfy Eq. (21.39), will also be stationary points of the Lyapunov function $V(\mathbf{a})$.

Effect of Gain

The Hopfield Lyapunov function can be simplified if we consider those cases where the amplifier gain γ is large. Recall that the nonlinear amplifier characteristic for our previous example was

$$a = f(n) = \frac{2}{\pi}\tan^{-1}\left(\frac{\gamma\pi n}{2}\right). \tag{21.48}$$

This function is displayed in Figure 21.7 for four different gain values.

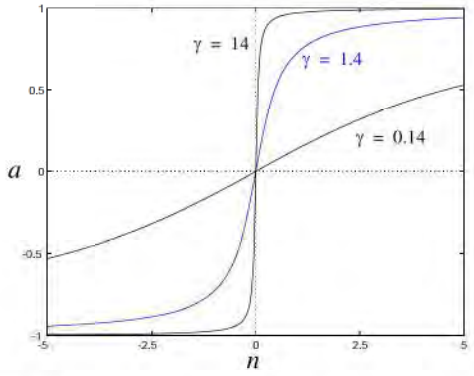


Figure 21.7 Inverse Tangent Amplifier Characteristic

The gain γ determines the steepness of the curve at $n = 0$. As γ increases, the slope of the curve at the origin increases. As γ goes to infinity, $f(n)$ approaches a signum (step) function.

Now recall from Eq. (21.8) that the general Lyapunov function is

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} + \sum_{i=1}^{S}\left\{\int_0^{a_i} f^{-1}(u)du\right\} - \mathbf{b}^T\mathbf{a}. \tag{21.49}$$

For our previous example,

$$f^{-1}(u) = \frac{2}{\gamma\pi}\tan\left(\frac{\pi u}{2}\right). \tag{21.50}$$

Therefore, the second term in the Lyapunov function takes the form

$$\int_0^{a_i} f^{-1}(u)du = \frac{2}{\gamma\pi}\left[\frac{2}{\pi}\log\left(\cos\left(\frac{\pi a_i}{2}\right)\right)\right] = -\frac{4}{\gamma\pi^2}\log\left[\cos\left(\frac{\pi a_i}{2}\right)\right]. \tag{21.51}$$

A graph of this function is shown in Figure 21.8 for three different values of the gain. Note that as γ increases the function flattens and is close to 0 most of the time. Thus, as the gain γ goes to infinity, the integral in the second term of the Lyapunov function will be close to zero in the range $-1 < a_i < 1$. This allows us to eliminate that term, and the *high-gain Lyapunov function* then reduces to

High-Gain Lyapunov Function

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} - \mathbf{b}^T\mathbf{a}. \tag{21.52}$$

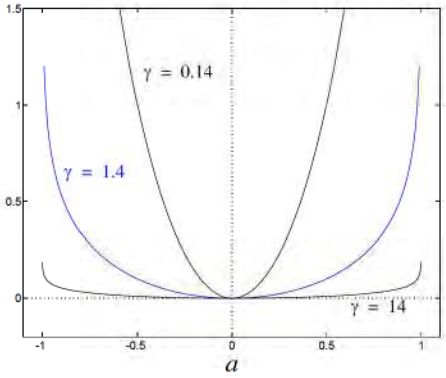


Figure 21.8 Second Term in the Lyapunov Function

By comparing Eq. (21.52) with Eq. (8.35), we can see that the high-gain Lyapunov function is, in fact, a quadratic function:

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} - \mathbf{b}^T\mathbf{a} = \frac{1}{2}\mathbf{a}^T\mathbf{A}\mathbf{a} + \mathbf{d}^T\mathbf{a} + c, \tag{21.53}$$

where

$$\nabla^2 V(\mathbf{a}) = \mathbf{A} = -\mathbf{W}, \ \mathbf{d} = -\mathbf{b} \text{ and } c = 0. \tag{21.54}$$

This is an important development, for now we can apply our results from Chapter 8 on quadratic functions to the understanding of the operation of Hopfield networks.

Recall that the shape of the surface of a quadratic function is determined by the eigenvalues and eigenvectors of its Hessian matrix. The Hessian matrix for our example Lyapunov function is

$$\nabla^2 V(\mathbf{a}) = -\mathbf{W} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}. \tag{21.55}$$

The eigenvalues of this Hessian matrix are computed as follows:

$$\left|\nabla^2 V(\mathbf{a}) - \lambda \mathbf{I}\right| = \begin{vmatrix} -\lambda & -1 \\ -1 & -\lambda \end{vmatrix} = \lambda^2 - 1 = (\lambda + 1)(\lambda - 1). \tag{21.56}$$

Thus, the eigenvalues are $\lambda_1 = -1$ and $\lambda_2 = 1$. It follows that the eigenvectors are

$$\mathbf{z}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{z}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}. \tag{21.57}$$

What does the surface of the high-gain Lyapunov function look like? We know, since the Hessian matrix has one positive and one negative eigenvalue, that we have a saddle point condition. The surface will have a negative curvature along the first eigenvector and a positive curvature along the second eigenvector. The surface is shown in Figure 21.9.

The function does not have a minimum. However, the network is constrained to the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$ by the amplifier transfer function. Therefore, there will be constrained minima at the two corners of the hypercube

$$\mathbf{a} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{a} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}. \tag{21.58}$$

When the gain is very small, there is a single minimum at the origin (see Exercise E21.1). As the gain is increased, two minima move out from the origin toward the two corners given by Eq. (21.58). Figure 21.3 displays an intermediate case, where the gain is $\gamma = 1.4$. The minima in that figure occur at

$$\mathbf{a} = \begin{bmatrix} 0.57 \\ 0.57 \end{bmatrix} \text{ and } \mathbf{a} = \begin{bmatrix} -0.57 \\ -0.57 \end{bmatrix}. \tag{21.59}$$

In the general case, where there are more than two neurons in the network, the high-gain minima will fall in certain corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$. We will discuss the general case in more detail in later sections, after we describe the Hopfield design process.

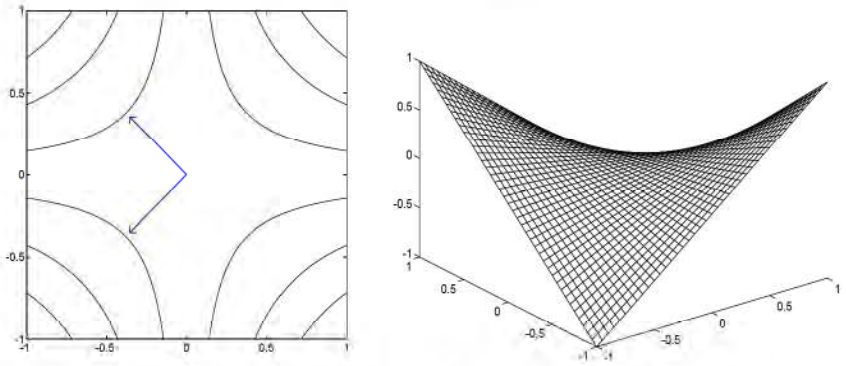


Figure 21.9 Example High Gain Lyapunov Function

Hopfield Design

The Hopfield network does not have a learning law associated with it. It is not trained, nor does it learn on its own. Instead, a design procedure based on the Lyapunov function is used to determine the weight matrix.

Consider again the high-gain Lyapunov function

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} - \mathbf{b}^T\mathbf{a}. \tag{21.60}$$

The Hopfield design technique is to choose the weight matrix $\mathbf{W}$ and the bias vector $\mathbf{b}$ so that V takes on the form of a function that you want to minimize. Convert whatever problem you want to solve into a quadratic minimization problem. Since the Hopfield network will minimize V, it will also solve the original problem. The trick, of course, is in the conversion, which is generally not straightforward.

Content-Addressable Memory

Content-Addressable Memory

In this section we will describe how a Hopfield network can be designed to work as an associative memory. The type of associative memory we will design is called a *content-addressable memory*, because it retrieves stored memories on the basis of part of the contents. This is in contrast to standard computer memories, where items are retrieved based on their addresses. For example, if we have a content-addressable data base that contains names, addresses and phone numbers of employees, we can retrieve a complete data entry simply by providing the employee name (or

perhaps a partial name). The content-addressable memory is effectively the same as the autoassociative memory described in Chapter 7 (see page 7-10), except that in this chapter we will use the recurrent Hopfield network instead of the linear associator.

Suppose that we want to store a set of prototype patterns in a Hopfield network. When an input pattern is presented to the network, the network should produce the stored pattern that most closely resembles the input pattern. The initial network output is assigned to the input pattern. The network output should then converge to the prototype pattern closest to the input pattern. For this to happen, the prototype patterns must be minima of the Lyapunov function.

Let's assume that the prototype patterns are

$$\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\}. \tag{21.61}$$

Each of these vectors consists of S elements, having the values 1 or -1. Assume further that $Q \ll S$, so that the state space is large, and that the prototype patterns are well distributed in this space, and so will not be close to each other.

In order for a Hopfield network to be able to recall the prototype patterns, the patterns must be minima of the Lyapunov function. Since the high-gain Lyapunov function is quadratic, we need the prototype patterns to be (constrained) minima of an appropriate quadratic function. We propose the following quadratic performance index:

$$J(\mathbf{a}) = -\frac{1}{2}\sum_{q=1}^{Q}([\mathbf{p}_q]^T\mathbf{a})^2. \tag{21.62}$$

If the elements of the vectors $\mathbf{a}$ are restricted to be ± 1, this function is minimized at the prototype patterns, as we will now show.

Assume that the prototype patterns are orthogonal. If we evaluate the performance index at one of the prototype patterns, we find

$$J(\mathbf{p}_j) = -\frac{1}{2}\sum_{q=1}^{Q}([\mathbf{p}_q]^T\mathbf{p}_j)^2 = -\frac{1}{2}([\mathbf{p}_j]^T\mathbf{p}_j)^2 = -\frac{S}{2}. \tag{21.63}$$

The second equality follows from the orthogonality of the prototype patterns. The last equality follows because all elements of $\mathbf{p}_j$ are ± 1.

Next, evaluate the performance index at a random input pattern $\mathbf{a}$, which is presumably not close to any prototype pattern. Each element in the sum in Eq. (21.62) is an inner product between a prototype pattern and the input. The inner product will increase as the input moves closer to a proto-

type pattern. However, if the input is not close to any prototype pattern, then all terms of the sum in Eq. (21.62) will be small. Therefore, $J(\mathbf{a})$ will be largest (least negative) when $\mathbf{a}$ is not close to any prototype pattern, and will be smallest (most negative) when $\mathbf{a}$ is equal to any one of the prototype patterns.

We have now found a quadratic function that accurately indicates the performance of the content-addressable memory. The next step is to choose the weight matrix $\mathbf{W}$ and bias $\mathbf{b}$ so that the Hopfield Lyapunov function V will be equivalent to the quadratic performance index J.

If we use the supervised Hebb rule to compute the weight matrix (with target patterns being the same as input patterns) as

$$\mathbf{W} = \sum_{q=1}^{Q} \mathbf{p}_q(\mathbf{p}_q)^T, \tag{21.64}$$

and set the bias to zero

$$\mathbf{b} = \mathbf{0}, \tag{21.65}$$

then the Lyapunov function is

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} = -\frac{1}{2}\mathbf{a}^T\left[\sum_{q=1}^{Q} \mathbf{p}_q(\mathbf{p}_q)^T\right]\mathbf{a} = -\frac{1}{2}\sum_{q=1}^{Q} \mathbf{a}^T\mathbf{p}_q(\mathbf{p}_q)^T\mathbf{a}. \tag{21.66}$$

This can be rewritten

$$V(\mathbf{a}) = -\frac{1}{2}\sum_{q=1}^{Q} [(\mathbf{p}_q)^T\mathbf{a}]^2 = J(\mathbf{a}). \tag{21.67}$$

Therefore, the Lyapunov function is indeed equal to the quadratic performance index for the content-addressable memory problem. The Hopfield network output will tend to converge to the stored prototype patterns (among other possible equilibrium points, as we will discuss later).

As noted in Chapter 7, the supervised Hebb rule does not work well if there is significant correlation between the prototype patterns. In that case the pseudoinverse technique has been suggested. Another design technique, which is beyond the scope of this text, is given in [LiMi89].

In the best situation, where the prototype patterns are orthogonal, every prototype pattern will be an equilibrium point of the network. However, there will be many other equilibrium points as well. The network may well converge to a pattern that is not one of the prototype patterns. A general rule is that, when using the Hebb rule, the number of stored patterns can

be no more than 15% of the number of neurons.The reference [LiMi89] discusses more complex design procedures, which minimize the number of spurious equilibrium points.

In the next section we will analyze the location of the equilibrium points more closely.

Hebb Rule

Let's take a closer look at the operation of the Hopfield network when the Hebb rule is used to compute the weight matrix and the prototype patterns are orthogonal. (The following analysis follows the discussion in the Chapter 7, Problem P7.5.) The supervised Hebb rule is given by

$$\mathbf{W} = \sum_{q=1}^{Q} \mathbf{p}_q (\mathbf{p}_q)^T . \tag{21.68}$$

If we apply the prototype vector $\mathbf{p}_j$ to the network, then

$$\mathbf{W}\mathbf{p}_j = \sum_{q=1}^{Q} \mathbf{p}_q (\mathbf{p}_q)^T \mathbf{p}_j = \mathbf{p}_j (\mathbf{p}_j)^T \mathbf{p}_j = S\mathbf{p}_j , \tag{21.69}$$

where the second equality holds because the prototype patterns are orthogonal, and the third equality holds because each element of $\mathbf{p}_j$ is either 1 or -1. Eq. (21.69) is of the form

$$\mathbf{W}\mathbf{p}_j = \lambda \mathbf{p}_j . \tag{21.70}$$

Therefore, each prototype vector is an eigenvector of the weight matrix and they have a common eigenvalue of $\lambda = S$. The eigenspace X for the eigenvalue $\lambda = S$ is therefore

$$X = \text{span}\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\} . \tag{21.71}$$

This space contains all vectors that can be written as linear combinations of the prototype vectors. That is, any vector $\mathbf{a}$ that is a linear combination of the prototype vectors is an eigenvector.

$$\begin{aligned} \mathbf{W}\mathbf{a} &= \mathbf{W}\{\alpha_1\mathbf{p}_1 + \alpha_2\mathbf{p}_2 + \cdots + \alpha_Q\mathbf{p}_Q\} \\ &= \{\alpha_1\mathbf{W}\mathbf{p}_1 + \alpha_2\mathbf{W}\mathbf{p}_2 + \cdots + \alpha_Q\mathbf{W}\mathbf{p}_Q\} \\ &= \{\alpha_1 S\mathbf{p}_1 + \alpha_2 S\mathbf{p}_2 + \cdots + \alpha_Q S\mathbf{p}_Q\} \\ &= S\{\alpha_1\mathbf{p}_1 + \alpha_2\mathbf{p}_2 + \cdots + \alpha_Q\mathbf{p}_Q\} = S\mathbf{a} \end{aligned} \tag{21.72}$$

The eigenspace for the eigenvalue $\lambda = S$ is Q-dimensional (assuming that the prototype vectors are independent).

The entire space R^S can be divided into two disjoint sets [Brog85],

$$R^S = X \cup X^{\perp}, \tag{21.73}$$

where $X^{\perp}$ is the orthogonal complement of X. (This is true for any set X, not just the one we are considering here.) Every vector in $X^{\perp}$ is orthogonal to every vector in X. This means that for any vector $\mathbf{a} \in X^{\perp}$,

$$(\mathbf{p}_q)^T \mathbf{a} = 0, \quad q = 1, 2, \ldots, Q. \tag{21.74}$$

Therefore, if $\mathbf{a} \in X^{\perp}$,

$$\mathbf{W}\mathbf{a} = \sum_{q=1}^{Q} \mathbf{p}_q (\mathbf{p}_q)^T \mathbf{a} = \sum_{q=1}^{Q} (\mathbf{p}_q \cdot 0) = \mathbf{0} = 0 \cdot \mathbf{a}. \tag{21.75}$$

So $X^{\perp}$ defines an eigenspace for the repeated eigenvalue $\lambda = 0$.

To summarize, the weight matrix has two eigenvalues, S and 0. The eigenspace for the eigenvalue S is the space spanned by the prototype vectors. The eigenspace for the eigenvalue 0 is the orthogonal complement of the space spanned by the prototype vectors.

Since (from Eq. (21.54)) the Hessian matrix for the high-gain Lyapunov function V is

$$\nabla^2 V = -\mathbf{W}, \tag{21.76}$$

the eigenvalues for $\nabla^2 V$ will be $-S$ and 0.

The high-gain Lyapunov function is a quadratic function. Therefore, the eigenvalues of the Hessian matrix determine its shape. Because the first eigenvalue is negative, V will have negative curvature in X. Because the second eigenvalue is zero, V will have zero curvature in $X^{\perp}$.

What do these results say about the response of the Hopfield network? Because V has negative curvature in X, the trajectories of the Hopfield network will tend to fall into the corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$ that are contained in X.

Note that if we compute the weight matrix using the Hebb rule, there will be at least two minima of the Lyapunov function for each prototype vector. If $\mathbf{p}_q$ is a prototype vector, then $-\mathbf{p}_q$ will also be in the space spanned by the prototype vectors, X. Therefore, the negative of each prototype vector will be one of the corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$ that are con-

tained in X. There will also be a number of other minima of the Lyapunov function that do not correspond to prototype patterns.

Spurious Patterns

The minima of V are in the corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$ that are contained in X. These corners will include the prototype patterns, but they will also include some linear combinations of the prototype patterns. Those minima that are not prototype patterns are often referred to as *spurious patterns*. The objective of Hopfield network design is to minimize the number of spurious patterns and to make the basins of attraction for each of the prototype patterns as large as possible. A design method that is guaranteed to minimize the number of spurious patterns is described in [LiMi89].



To illustrate these principles, consider again the second-order example we have been discussing, where the connection matrix is

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{21.77}$$

Suppose that this had been designed using the Hebb rule with one prototype pattern (obviously not an interesting practical case):

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{21.78}$$

Then

$$\mathbf{W} = \mathbf{p}_1(\mathbf{p}_1)^T = \begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \tag{21.79}$$

Notice that

$$\mathbf{W}' = \mathbf{W} - \mathbf{I} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{21.80}$$

corresponds to our original connection matrix. More about this in the next section.

The high-gain Lyapunov function is

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} = -\frac{1}{2}\mathbf{a}^T\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\mathbf{a}. \tag{21.81}$$

The Hessian matrix for $V(\mathbf{a})$ is

$$\nabla^2 V(\mathbf{a}) = -\mathbf{W} = \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}. \tag{21.82}$$

Its eigenvalues are

$$\lambda_1 = -S = -2, \text{ and } \lambda_2 = 0, \tag{21.83}$$

and the corresponding eigenvectors are

$$\mathbf{z}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{z}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}. \tag{21.84}$$

The first eigenvector, corresponding to the eigenvalue $-S$, represents the space spanned by the prototype vector:

$$X = \{\mathbf{a}: a_1 = a_2\}. \tag{21.85}$$

The second eigenvector, corresponding to the eigenvalue 0, represents the orthogonal complement of the first eigenvector:

$$X^{\perp} = \{\mathbf{a}: a_1 = -a_2\}. \tag{21.86}$$

The Lyapunov function is displayed in Figure 21.10.

This surface has a straight ridge from the upper-left to the lower-right corner. This represents the zero curvature region of $X^{\perp}$. Initial conditions to the left or to the right of the ridge will converge to the points

$$\mathbf{a} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ or } \mathbf{a} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \tag{21.87}$$

respectively. Initial conditions exactly on this ridge will stabilize where they start. This situation is the same as that for our original example (see Figure 21.9), except that in that case, initial points on the sloping ridge converged to the origin, instead of remaining where they started (see Figure 21.6). Initial points to the right or to the left of the ridge, in both systems, converge to the prototype design points. Thus, the convergence of our original system, and the convergence of the system with zero diagonal elements, are identical in every important aspect. We will investigate this further in the next section.

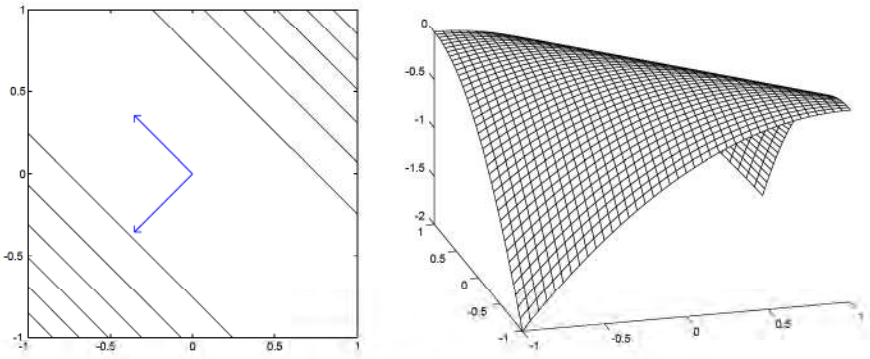


Figure 21.10 Example Lyapunov Function

Lyapunov Surface

In many discussions of the Hopfield network the diagonal elements of the weight matrix are set to zero. In this section we will analyze the effect of this operation on the Lyapunov surface. (See also Chapter 7, Exercise E7.5.)

For the content-addressable memory network, all of the diagonal elements of the weight matrix will be equal to Q (the number of prototype patterns), since the elements of each $\mathbf{p}_q$ are ± 1. Therefore, we can zero the diagonal by subtracting Q times the identity matrix:

$$\mathbf{W}' = \mathbf{W} - Q\mathbf{I}. \tag{21.88}$$

Let's investigate how this change affects the form of the Lyapunov function. If we multiply this new weight matrix times one of the prototype vectors we find

$$\mathbf{W}'\mathbf{p}_q = [\mathbf{W} - Q\mathbf{I}]\mathbf{p}_q = S\mathbf{p}_q - Q\mathbf{p}_q = (S - Q)\mathbf{p}_q. \tag{21.89}$$

Therefore, $(S - Q)$ is an eigenvalue of $\mathbf{W}'$, and the corresponding eigenspace is X, the space spanned by the prototype vectors.

If we multiply the new weight matrix times a vector from the orthogonal complement space, $\mathbf{a} \in X^{\perp}$, we find

$$\mathbf{W}'\mathbf{a} = [\mathbf{W} - Q\mathbf{I}]\mathbf{a} = \mathbf{0} - Q\mathbf{a} = -Q\mathbf{a}. \tag{21.90}$$

Therefore, $-Q$ is an eigenvalue of $\mathbf{W}'$, and the corresponding eigenspace is $X^{\perp}$.

To summarize, the eigenvectors of $\mathbf{W}'$ are the same as the eigenvectors of $\mathbf{W}$, but the eigenvalues are now $(S - Q)$ and $-Q$, instead of S and 0. There-

fore, the eigenvalues of the Hessian matrix of the modified Lyapunov function, $\nabla^2 V'(\mathbf{a}) = -\mathbf{W}'$, are $-(S-Q)$ and Q.

This implies that the energy surface will have negative curvature in X and positive curvature in $X^{\perp}$, in contrast with the original Lyapunov function, which had negative curvature in X and zero curvature in $X^{\perp}$.

A comparison of Figure 21.9 and Figure 21.10 demonstrates the effect on the Lyapunov function of setting the diagonal elements of the weight matrix to zero. In terms of system performance, the change has little effect. If the initial condition of the Hopfield network falls anywhere off of the line $a_1 = -a_2$, then, in either case, the output of the network will converge to one of the corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$, which consists of the two points $\mathbf{a} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$ and $\mathbf{a} = \begin{bmatrix} -1 & -1 \end{bmatrix}^T$.

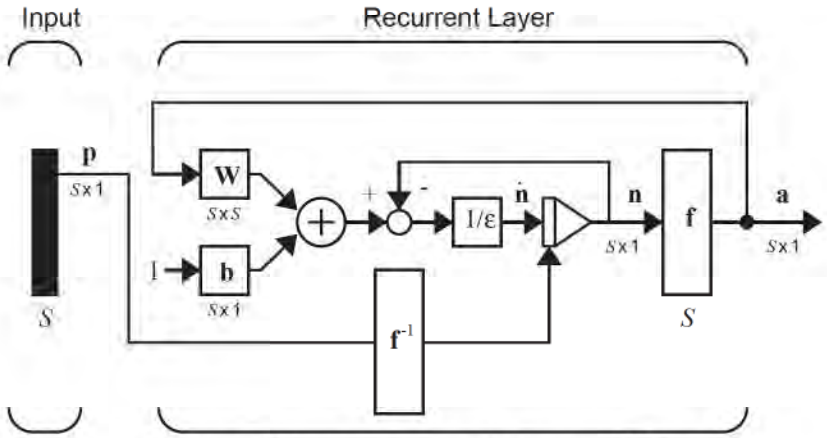
If the initial condition falls exactly on the line $a_1 = -a_2$, and the weight matrix $\mathbf{W}$ is used, then the network output will remain constant. If the initial condition falls exactly on the line $a_1 = -a_2$, and the weight matrix $\mathbf{W}'$ is used, then the network output will converge to the saddle point at the origin (as in Figure 21.6). Neither of these results is desirable, since the network output does not converge to a minimum of the Lyapunov function. Of course, the only case in which the network converges to a saddle point is when the initial condition falls exactly on the line $a_1 = -a_2$, which would be highly unlikely in practice.

Summary of Results

Hopfield Model

$$\varepsilon \frac{d\mathbf{n}(t)}{dt} = -\mathbf{n}(t) + \mathbf{W}\mathbf{a}(t) + \mathbf{b}$$

$$\mathbf{a}(t) = \mathbf{f}(\mathbf{n}(t))$$



$$\mathbf{n}(0) = \mathbf{f}^{-1}(\mathbf{p}),\ (\mathbf{a}(0) = \mathbf{p}) \qquad \varepsilon\, d\mathbf{n}/dt = -\mathbf{n} + \mathbf{W}\mathbf{f}(\mathbf{n}) + \mathbf{b}$$

Lyapunov Function

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} + \sum_{i=1}^{S}\left\{\int_0^{a_i} f^{-1}(u)\,du\right\} - \mathbf{b}^T\mathbf{a}$$

$$\frac{d}{dt}V(\mathbf{a}) = -\varepsilon\sum_{i=1}^{S}\left(\frac{d}{da_i}[f^{-1}(a_i)]\right)\left(\frac{da_i}{dt}\right)^2$$

If $\frac{d}{da_i}[f^{-1}(a_i)] > 0$, then $\frac{d}{dt}V(\mathbf{a}) \le 0$.

Invariant Sets

The Invariant Set Consists of the Equilibrium Points.

$$L = Z = \{\mathbf{a}: d\mathbf{a}/dt = \mathbf{0},\ \mathbf{a} \text{ in the closure of } G\}$$

Hopfield Attractors

The Equilibrium Points Are Stationary Points.

$$\text{If } \frac{d\mathbf{a}(t)}{dt} = \mathbf{0}, \text{ then } \nabla V(\mathbf{a}) = \mathbf{0}.$$

$$\nabla V(\mathbf{a}) = [-\mathbf{W}\mathbf{a} + \mathbf{n} - \mathbf{b}] = -\varepsilon\left[\frac{d\mathbf{n}(t)}{dt}\right]$$

High-Gain Lyapunov Function

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} - \mathbf{b}^T\mathbf{a}$$

$$\nabla^2 V(\mathbf{a}) = -\mathbf{W}$$

Content-Addressable Memory

$$\mathbf{W} = \sum_{q=1}^{Q} \mathbf{p}_q(\mathbf{p}_q)^T \text{ and } \mathbf{b} = \mathbf{0}$$

Energy Surface (Orthogonal Prototype Patterns)

Eigenvalues/Eigenvectors of $\nabla^2 V(\mathbf{a}) = -\mathbf{W}$ Are

$\lambda_1 = -S$, with eigenspace $X = \text{span}\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\}$.

$\lambda_2 = 0$, with eigenspace $X^\perp$.

$X^\perp$ is defined such that for any vector $\mathbf{a} \in X^\perp$, $(\mathbf{p}_q)^T\mathbf{a} = 0$, $q = 1, 2, \ldots, Q$

Trajectories (Orthogonal Prototype Patterns)

Because the first eigenvalue is negative, $V(\mathbf{a})$ will have negative curvature in X. Because the second eigenvalue is zero, $V(\mathbf{a})$ will have zero curvature in $X^\perp$. Because $V(\mathbf{a})$ has negative curvature in X, the trajectories of the Hopfield network will tend to fall into the corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$ that are contained in X.

Solved Problems

P21.1 Assume the binary prototype vectors

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}, \qquad \mathbf{p}_2 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}.$$

i. Design a continuous-time Hopfield network (specify connection weights) to recognize these patterns, using the Hebb rule.

ii. Find the Hessian matrix of the high-gain Lyapunov function for this network. What are the eigenvalues and eigenvectors of the Hessian matrix?

iii. Assuming large gain, what are the stable equilibrium points for this Hopfield network?

i. First calculate the weight matrix from the reference vectors, using the supervised Hebb rule.

$$\mathbf{W} = \mathbf{p}_1(\mathbf{p}_1)^T + \mathbf{p}_2(\mathbf{p}_2)^T = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix},$$

which simplifies to

$$\mathbf{W} = \begin{bmatrix} 2 & 0 & 0 & -2 \\ 0 & 2 & -2 & 0 \\ 0 & -2 & 2 & 0 \\ -2 & 0 & 0 & 2 \end{bmatrix}.$$

ii. The Hessian of the high-gain Lyapunov function, from Eq. (21.54), is the negative of the weight matrix:

$$\nabla^2 V(\mathbf{a}) = \begin{bmatrix} -2 & 0 & 0 & 2 \\ 0 & -2 & 2 & 0 \\ 0 & 2 & -2 & 0 \\ 2 & 0 & 0 & -2 \end{bmatrix}.$$

The prototype patterns are orthogonal ($[\mathbf{p}_1]^T\mathbf{p}_2 = 0$). Thus, the eigenvalues are $\lambda_1 = -S = -4$ and $\lambda_2 = 0$. The eigenspace for $\lambda_1 = -4$ is

$$X = \text{span}\{\mathbf{p}_1, \mathbf{p}_2\}.$$

The eigenspace for $\lambda_2 = 0$ is the orthogonal complement of X:

$$X^{\perp} = \text{span}\left\{\begin{bmatrix}1\\1\\1\\1\end{bmatrix}, \begin{bmatrix}1\\-1\\-1\\1\end{bmatrix}\right\},$$

where we have chosen two vectors that are orthogonal to both $\mathbf{p}_1$ and $\mathbf{p}_2$.

iii. The stable points will be $\mathbf{p}_1, \mathbf{p}_2, -\mathbf{p}_1, -\mathbf{p}_2$ since the negative of the prototype patterns will also be equilibrium points. There may be other equilibrium points, if other corners of the hypercube lie in the span $\{\mathbf{p}_1, \mathbf{p}_2\}$. There are a total of $2^4 = 16$ corners of the hypercube. Four will fall in X and four will fall in $X^{\perp}$. The other corners are partly in X and partly in X^T.

P21.2 Consider a high-gain Hopfield network with a weight matrix and bias given by

$$\mathbf{W} = \begin{bmatrix}-1 & -1\\-1 & -1\end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix}1\\-1\end{bmatrix}.$$

i. Sketch a contour plot of the high-gain Lyapunov function for this network.

ii. If the network is given the initial condition $\begin{bmatrix}1 & 1\end{bmatrix}^T$, where will the network converge?

i. First consider the high-gain Lyapunov function

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T\mathbf{W}\mathbf{a} - \mathbf{b}^T\mathbf{a}.$$

The Hessian matrix is

$$\nabla^2 V(\mathbf{a}) = -\mathbf{W} = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}.$$

Next, we need to compute the eigenvalues and eigenvectors:

$$|\nabla^2 V(\mathbf{a}) - \lambda \mathbf{I}| = \left| \begin{bmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{bmatrix} \right| = \lambda^2 - 2\lambda + 1 - 1 = \lambda(\lambda - 2).$$

The eigenvalues are $\lambda_1 = 0$ and $\lambda_2 = 2$.

Now we can find the eigenvectors. For $\lambda_1 = 0$,

$$[\nabla^2 V(\mathbf{a}) - \lambda_1 \mathbf{I}]\mathbf{z}_1 = \mathbf{0},$$

and therefore

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \mathbf{z}_1 = \mathbf{0} \text{ or } \mathbf{z}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Similarly, for $\lambda_2 = 2$,

$$[\nabla^2 V(\mathbf{a}) - \lambda_2 \mathbf{I}]\mathbf{z}_2 = \mathbf{0}$$

and therefore

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \mathbf{z}_2 = \mathbf{0} \text{ or } \mathbf{z}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

So the term

$$-\frac{1}{2}\mathbf{a}^T \mathbf{W} \mathbf{a}$$

has zero curvature in the direction $\mathbf{z}_1$ and positive curvature in the direction $\mathbf{z}_2$.

Now we have to account for the linear term. First plot the contour without the linear term, as in Figure P21.1.

The linear term will cause a negative slope in the direction of

$$\mathbf{b} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Therefore everything will curve down toward $\begin{bmatrix} 1 & -1 \end{bmatrix}^T$, as is shown in Figure P21.2.

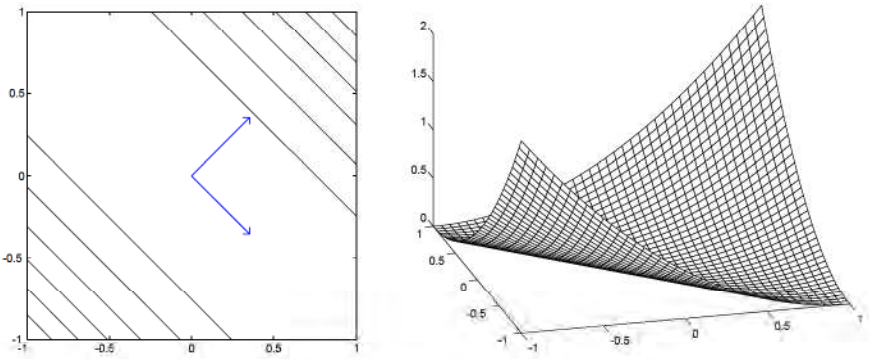


Figure P21.1 Contour Without Linear Term

ii. All trajectories will converge to $\begin{bmatrix} 1 & -1 \end{bmatrix}^T$, regardless of the initial conditions. As we can see in Figure P21.2, the energy function has only one minimum, which is located at $\begin{bmatrix} 1 & -1 \end{bmatrix}^T$. (Keep in mind that the output of the network is constrained to fall within the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$.)

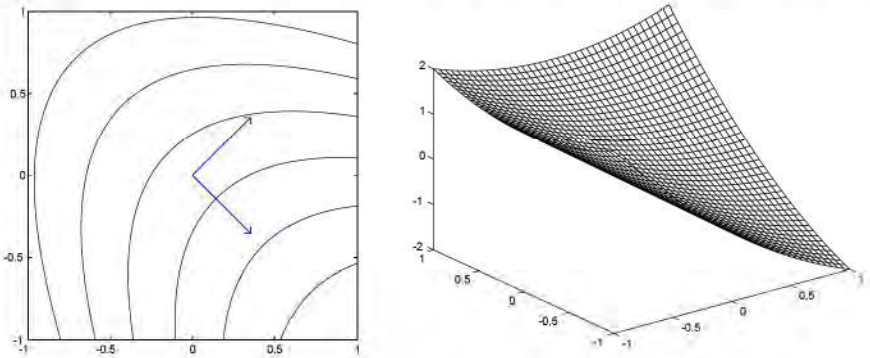


Figure P21.2 Contour Including Linear Term

P21.3 Consider the following prototype vectors.

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

i. Design a Hopfield network to recognize these patterns.

ii. Find the Hessian matrix of the high-gain Lyapunov function for this network. What are the eigenvalues and eigenvectors of the Hessian matrix?

iii. What are the stable points for this Hopfield network (assume large gain)? What are the basins of attraction?

iv. How well does this network perform the pattern recognition problem?

i. We will use the Hebb rule to find the weight matrix.

$$\mathbf{W} = \mathbf{p}_1(\mathbf{p}_1)^T + \mathbf{p}_2(\mathbf{p}_2)^T = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

The bias is set to zero.

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

ii. The Hessian matrix of the high-gain Lyapunov function is the negative of the weight matrix.

$$\nabla^2 V(\mathbf{a}) = -\mathbf{W} = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix}$$

By inspection, we can see that there is a repeated eigenvalue.

$$\lambda_1 = \lambda_2 = -S = -2$$

The eigenvectors will then be

$$\mathbf{z}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } \mathbf{z}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

or any linear combination. (The entire $\Re^2$ is the eigenspace for the eigenvalue $\lambda = -2$.)

iii. From Chapter 8 we know that when the eigenvalues of the Hessian are equal, the contours will be circular. Because the eigenvalues are negative, the function will have a single maximum at the origin. There will be four minima at the four corners of the hypercube $\{\mathbf{a}: -1 < a_i < 1\}$. There are also four saddle points. The high-gain Lyapunov function is displayed in Figure P21.3.

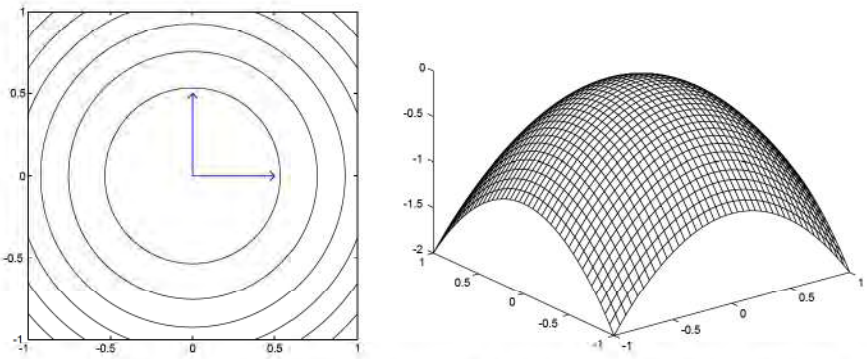


Figure P21.3 High-Gain Lyapunov Function for Problem P21.3

There are a total of nine stationary points. We could use the corollary to La-Salle's Invariance Theorem to show that the maximum point at the origin has a basin of attraction that only includes the origin itself. Therefore it is not a stable equilibrium point. The saddle points have regions of attraction that are lines. (For example, the saddle point at $\begin{bmatrix} -1 & 0 \end{bmatrix}^T$ has a region of attraction along the negative a_1 axis.) The four corners of the hypercube are the only attractors that have two-dimensional regions of attraction. The region of attraction for each corner is the corresponding quadrant of the hypercube. Figure P21.4 shows the low-gain Lyapunov function (with gain $\gamma = 1.4$) and illustrates convergence to a saddle point and to a minimum.

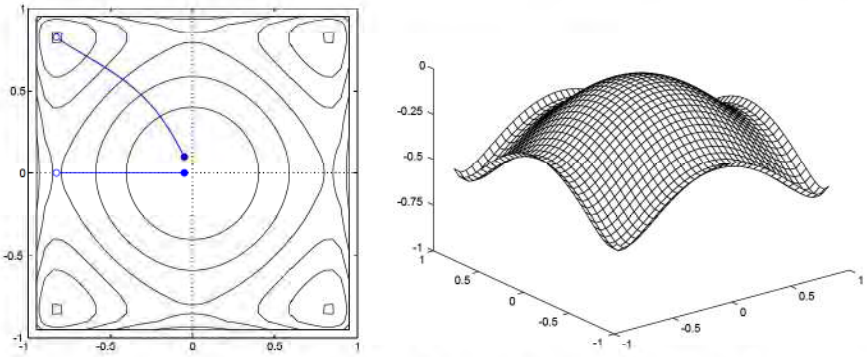


Figure P21.4 Lyapunov Function for Problem P21.3

iv. The network does not do a good job on the pattern recognition problem. Not only does it recognize the two prototype patterns, but it also "recognizes" the other two corners of the hypercube as well. The network will converge to whichever corner is closest to the input pattern, even though we only wanted it to store the two prototype patterns. Since every possible

two-bit pattern has been stored, the network is not very useful. This is not unexpected, since the number of patterns that the Hebb rule is expected to store is only 15% of the number of neurons. Since we only have two neurons, we can't expect to successfully store many patterns. Try Exercise E21.2 for a better network.

P21.4 A Hopfield network has the following high-gain Lyapunov function:

$$V(\mathbf{a}) = -\frac{1}{2}(7(a_1)^2 + 12a_1a_2 - 2(a_2)^2).$$

i. Find the weight matrix.

ii. Find the gradient vector of the Lyapunov function.

iii. Find the Hessian matrix of the Lyapunov function.

iv. Sketch a contour plot of the Lyapunov function.

v. Sketch the path that a steepest descent algorithm would follow for $V(\mathbf{a})$ with an initial condition of $\begin{bmatrix}0.25 & 0.25\end{bmatrix}^T$.

i. $V(\mathbf{a})$ is a quadratic function, which can be rewritten as

$$V(\mathbf{a}) = -\frac{1}{2}(7(a_1)^2 + 12a_1a_2 - 2(a_2)^2) = -\frac{1}{2}\mathbf{a}^T\begin{bmatrix}7 & 6\\ 6 & -2\end{bmatrix}\mathbf{a}.$$

Therefore the weight matrix is

$$\mathbf{W} = \begin{bmatrix}7 & 6\\ 6 & -2\end{bmatrix}.$$

ii. Since $V(\mathbf{a})$ is a quadratic function, we can use Eq. (8.38) to find the gradient.

$$\nabla V(\mathbf{a}) = -\begin{bmatrix}7 & 6\\ 6 & -2\end{bmatrix}\mathbf{a}$$

iii. From Eq. (8.39), the Hessian is

$$\nabla^2 V(\mathbf{a}) = -\begin{bmatrix}7 & 6\\ 6 & -2\end{bmatrix} = \begin{bmatrix}-7 & -6\\ -6 & 2\end{bmatrix}.$$

iv. To compute the eigenvalues,

$$|\nabla^2 V(\mathbf{a}) - \lambda \mathbf{I}| = \left| \begin{bmatrix} -7-\lambda & -6 \\ -6 & 2-\lambda \end{bmatrix} \right| = \lambda^2 + 5\lambda - 50 = (\lambda + 10)(\lambda - 5).$$

The eigenvalues are $\lambda_1 = -10$ and $\lambda_2 = 5$.

Now we can find the eigenvectors. For $\lambda_1 = -10$,

$$[\nabla^2 V(\mathbf{a}) - \lambda_1 \mathbf{I}]\mathbf{z}_1 = \mathbf{0},$$

and therefore

$$\begin{bmatrix} 3 & -6 \\ -6 & 12 \end{bmatrix} \mathbf{z}_1 = \mathbf{0} \text{ or } \mathbf{z}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

Similarly, for $\lambda_2 = 5$,

$$[\nabla^2 V(\mathbf{a}) - \lambda_2 \mathbf{I}]\mathbf{z}_2 = \mathbf{0}$$

and therefore

$$\begin{bmatrix} -12 & -6 \\ -6 & -3 \end{bmatrix} \mathbf{z}_2 = \mathbf{0} \text{ or } \mathbf{z}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}.$$

Note that this is a saddle point case, since $\lambda_1 < 0 < \lambda_2$. There will be negative curvature along $\mathbf{z}_1$ and positive curvature along $\mathbf{z}_2$. The contour plot of the high-gain Lyapunov function is shown in Figure P21.5.

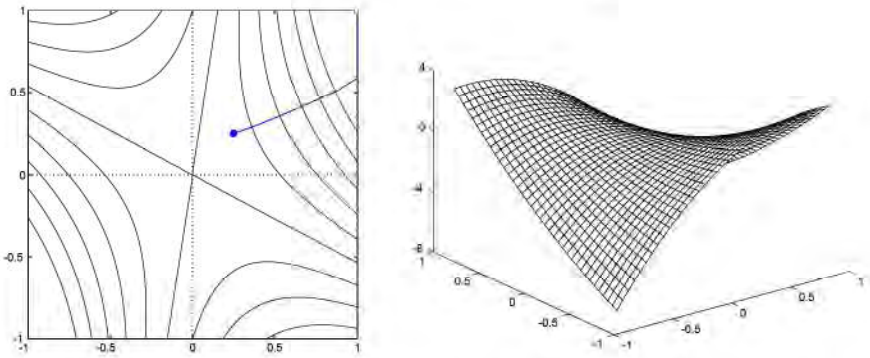


Figure P21.5 High-Gain Lyapunov Func. & Steepest Descent Trajectory

v. The steepest descent path will follow the negative of the gradient and will be perpendicular to the contour lines, as we saw in Chapter 9. When

the trajectory hits the edge of the hypercube, it follows the edge down to the minimum point. The resulting trajectory is shown in Figure P21.5.

The high-gain Lyapunov function is only an approximation, since it assumes infinite gain. As a comparison, Figure P21.6 illustrates the Lyapunov function, and the Hopfield trajectory, for a gain of 0.5.

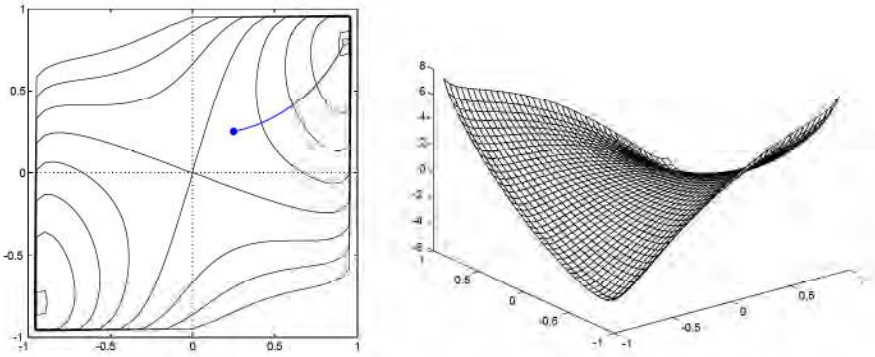


Figure P21.6 Lyapunov Function & Hopfield Trajectory

P21.5 The Hopfield network has been used for applications other than content-addressable memory. One of these other applications is analog-to-digital (A/D) conversion [HoTa86]. The function of the A/D converter is to take an analog signal y, and convert it into a series of bits (zeros and ones). For example, a two-bit A/D converter would try to approximate the signal y as follows:

$$y \cong \sum_{i=1}^{2} a_i 2^{(i-1)} = a_1 + a_2 2,$$

where a_1 and a_2 are allowed values of 0 or 1. (This A/D converter would approximate analog values in the range from 0 to 3, with a resolution of 1.) Tank and Hopfield suggest the following performance index for the A/D conversion process:

$$J(\mathbf{a}) = \frac{1}{2}\left(y - \sum_{i=1}^{2} a_i 2^{(i-1)}\right)^2 - \frac{1}{2}\left(\sum_{i=1}^{2} 2^{2(i-1)} a_i (a_i - 1)\right),$$

where the first term represents the A/D conversion error, and the second term forces a_1 and a_2 to take on values of 0 or 1.

Show that this performance index can be written as the Lyapunov function of a Hopfield network and define the appropriate weight matrix and bias vector.

The first step is to expand the terms of the performance index.

$$\left(y - \sum_{i=1}^{2} a_i 2^{(i-1)}\right)^2 = y^2 - 2y \sum_{i=1}^{2} a_i 2^{(i-1)} + \sum_{j=1}^{2} \sum_{i=1}^{2} a_i a_j 2^{(i-1)+(j-1)},$$

$$\left(\sum_{i=1}^{2} 2^{2(i-1)} a_i (a_i - 1)\right) = \sum_{i=1}^{2} (a_i)^2 2^{2(i-1)} - \sum_{i=1}^{2} a_i 2^{2(i-1)}$$

If we substitute these terms back into the performance index we find

$$J(\mathbf{a}) = \frac{1}{2}\left(y^2 + \sum_{j=1}^{2} \sum_{\substack{i=1 \\ i \neq j}}^{2} a_i a_j 2^{(i-1)+(j-1)} + \sum_{i=1}^{2} a_i (2^{2(i-1)} - 2^i y)\right).$$

The first term is not a function of $\mathbf{a}$. Therefore, it does not affect where the minima will occur, and we can ignore it.

We now want to show that this performance index takes the form of a high-gain Lyapunov function:

$$V(\mathbf{a}) = -\frac{1}{2}\mathbf{a}^T \mathbf{W} \mathbf{a} - \mathbf{b}^T \mathbf{a}.$$

This will be the case if

$$\mathbf{W} = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} y - \frac{1}{2} \\ 2y - 2 \end{bmatrix}.$$

In this Hopfield network, unlike the content-addressable memory, the input to the network is the scalar y, which is then used to compute the bias vector. In the content-addressable memory, the inputs to the network were vector patterns, which became the initial conditions on the network outputs.

Note that in this network the transfer function must limit the output to the range $0 < a < 1$. One transfer function that could be used is

$$f(n) = \frac{1}{(1 - e^{-\gamma n})}.$$

Epilogue

In this chapter we have introduced the Hopfield model, one of the most influential neural network architectures. One of the reasons that Hopfield was so influential was that he emphasized the practical considerations of the network. He described how the network could be implemented as an electrical circuit, and VLSI implementations of Hopfield-type networks were built at an early stage.

Hopfield also explained how the network could be used to solve practical problems in pattern recognition and optimization. Some of the applications that Hopfield proposed for his networks were: content-addressable memory [Hopf82], A/D conversion [TaHo86] and linear programming and optimization tasks, such as the traveling salesman problem [HoTa85].

One of Hopfield's key contributions was the application of Lyapunov stability theory to the analysis of his recurrent networks. He also showed that, for high-gain amplifiers, the Lyapunov function for his network was a quadratic function, which was minimized by the network. This led to several design procedures. The idea behind the development of the design techniques was to convert a given task into a quadratic minimization problem, which the network could then solve.

The Hopfield network is the last topic we will cover in any detail in this text. However, we have certainly not exhausted all of the important neural network architectures. In the next chapter we will give you some ideas about where to go next to explore the subject further.

Further Reading

[Ande72] J. Anderson, "A simple neural network generating an interactive memory," *Mathematical Biosciences*, vol. 14, pp. 197–220, 1972.

Anderson proposed a "linear associator" model for associative memory. The model was trained, using a generalization of the Hebb postulate, to learn an association between input and output vectors. The physiological plausibility of the network was emphasized. Kohonen published a closely related paper at the same time [Koho72], although the two researchers were working independently.

[AnSi77] J. A. Anderson, J. W. Silverstein, S. A. Ritz and R. S. Jones, "Distinctive features, categorical perception, and probability learning: Some applications of a neural model," *Psychological Review*, vol. 84, pp. 413–451, 1977.

This article describes the brain-state-in-a-box neural network model. It combines the linear associator network with recurrent connections to form a more powerful autoassociative system. It uses a nonlinear transfer function to contain the network output within a hypercube.

[CoGr83] M. A. Cohen and S. Grossberg, "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 13, no. 5, pp. 815–826, 1983.

Cohen and Grossberg apply LaSalle's Invariance Theorem to the analysis of the stability of competitive neural networks. The network description is very general, and the authors show how their analysis can be applied to many different types of recurrent neural networks.

[Gros67] S. Grossberg, "Nonlinear difference-differential equations in prediction and learning theory," *Proceedings of the National Academy of Sciences*, vol. 58, pp. 1329–1334, 1967.

This early work of Grossberg's discusses the storage of information in dynamically stable configurations.

[Hopf82] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational properties," *Proceedings of the National Academy of Sciences*, vol. 79, pp. 2554–2558, 1982.

This is the original Hopfield neural network paper, which signaled the resurgence of the field of neural networks. It describes a discrete-time network that behaves as a content-addressable memory. Hopfield demonstrates that the network evolves so as to minimize a specific Lyapunov function.

[Hopf84] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences*, vol. 81, pp. 3088–3092, 1984.

Hopfield demonstrates how an analog electrical circuit can function as a model for a large network of neurons with a graded response. The Lyapunov function for this network is derived and is used to design a network for use as a content-addressable memory.

[HoTa85] J. J. Hopfield and D. W. Tank, " 'Neural' computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–154, 1985.

This article describes the application of Hopfield networks to the solution of optimization problems. The traveling salesman problem, in which the length of a trip through a number of cities with only one visit to each city is minimized, is mapped onto a Hopfield network.

[Koho72] T. Kohonen, "Correlation matrix memories," *IEEE Transactions on Computers*, vol. 21, pp. 353–359, 1972.

Kohonen proposed a correlation matrix model for associative memory. The model was trained, using the outer product rule (also known as the Hebb rule), to learn an association between input and output vectors. The mathematical structure of the network was emphasized. Anderson published a closely related paper at the same time [Ande72], although the two researchers were working independently.

[LiMi89] J. Li, A. N. Michel and W. Porod, "Analysis and synthesis of a class of neural networks: Linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, pp. 1405–1422, November 1989.

This article investigates a class of neural networks described by first-order linear differential equations defined on a closed hypercube (Hopfield-like networks). Wanted and unwanted equilibrium points fall at the corners of the cube. The authors discuss design procedures that minimize the number of spurious equilibrium points.

[McPi43] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics.*, vol. 5, pp. 115–133, 1943.

This article introduces the first mathematical model of a neuron in which a weighted sum of input signals is compared to a threshold to determine whether or not the neuron fires.

[TaHo86] D. W. Tank and J. J. Hopfield, "Simple 'neural' optimization networks: An A/D converter, signal decision circuit and a linear programming circuit," *IEEE Transactions on Circuits and Systems*, vol. 33, no. 5, pp. 533–541, 1986.

The authors describe how Hopfield neural networks can be designed to solve certain optimization problems. In one example the Hopfield network implements an analog-to-digital conversion.

Exercises

E21.1 In the Hopfield network example starting on page 18-8 we used a gain of $\gamma = 1.4$. Figure 21.3 displays the Lyapunov function for that example. The high-gain Lyapunov function for the example is shown in Figure 21.9.

i. Show that the minima of the Lyapunov function for this example will be located at points where $n_1 = n_2 = f(n_1) = f(n_2)$. (Use Eq. (21.42) and set the gradient of $V(\mathbf{a})$ to zero.)

ii. Investigate the change in location of the minima as the gain is varied from $\gamma = 0.1$ to $\gamma = 10$.



iii. Sketch the contour plot for several different values of gain in this interval. You will probably need to use MATLAB for this.

E21.2 In Problem P21.3 we used the supervised Hebb rule to design a Hopfield network to recognize the following patterns:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

If we use another design rule [LiMi89], we find the following weight matrix and bias

$$\mathbf{W} = \begin{bmatrix} 1 & 0 \\ 0 & -10 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0 \\ 11 \end{bmatrix}.$$

i. Graph the contour plot for the high-gain Lyapunov function, if this weight matrix and bias are used.

ii. Discuss the difference between the performance of this Hopfield network and the one designed in Problem P21.3.



iii. Write a MATLAB M-file to simulate the Hopfield network. Use the **ode45** routine. Plot the responses of this network for several initial conditions.

E21.3 A Hopfield network has the following high-gain Lyapunov function:

$$V(\mathbf{a}) = -\frac{1}{2}((a_1)^2 + 2a_1a_2 + 4(a_2)^2 + 6a_1 + 10a_2).$$

i. Find the weight matrix and bias vector for this network.

ii. Find the gradient and Hessian for $V(\mathbf{a})$.

iii. Sketch a contour plot of $V(\mathbf{a})$.

iv. Find the stationary point(s) for $V(\mathbf{a})$. Use the corollary to LaSalle's Invariance Theorem to find as much information as you can about basins of attraction for any stable equilibrium points.

E21.4 In Problem P21.5 we demonstrated how a Hopfield network could be designed to operate as an A/D converter.

i. Sketch the contour plot of the high-gain Lyapunov function for the two-bit A/D converter network using an input value of $y = 0.5$. Locate the minimum points.

ii. Repeat part (i) for an input value of $y = 2.5$.

iii. Use the answers to parts (i) and (ii) to explain how the network will operate. Will the network perform the A/D conversion correctly?

E21.5 Assume the binary prototype vectors

$$\mathbf{p}_1 = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}^T, \mathbf{p}_2 = \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix}^T.$$

i. Design a continuous-time Hopfield network (specify connection weights and biases only) to recognize these patterns, using the Hebb rule.

ii. Find the Hessian matrix of the high-gain Lyapunov function for this network. What are the eigenvalues and eigenvectors of the Hessian matrix? (This requires very little computation.)

iii. Assuming large gain, what are the stable equilibrium points for this Hopfield network?

E21.6 Repeat Exercise E21.5 for the following prototype vectors.

i. $\mathbf{p}_1 = \begin{bmatrix} 1 & -1 & 1 & 1 \end{bmatrix}^T, \mathbf{p}_2 = \begin{bmatrix} 1 & 1 & 1 & -1 \end{bmatrix}^T.$

ii. $\mathbf{p}_1 = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}^T, \mathbf{p}_2 = \begin{bmatrix} 1 & 1 & -1 & -1 \end{bmatrix}^T.$

iii. $\mathbf{p}_1 = \begin{bmatrix} -1 & -1 & 1 & 1 & -1 & -1 \end{bmatrix}^T, \mathbf{p}_2 = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 \end{bmatrix}^T.$

E21.7 Consider a high-gain Hopfield network with weight matrix and bias given by:

$$\mathbf{W} = \begin{bmatrix} 1 & 3 \\ 3 & 9 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}.$$

i. Sketch a contour plot of the high-gain Lyapunov function for this network.

ii. If the network is given the following initial condition, where will the network converge?

$$\mathbf{a}(0) = \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix}$$

E21.8 Design a high-gain Hopfield network (give the weights and the biases) with only one stable equilibrium point:

$$\begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Explain your procedure, and show all steps. (Do not use the Hebb rule.)

E21.9 Consider a high-gain Hopfield network with weight matrix and bias given by:

$$\mathbf{W} = \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

i. Sketch a contour plot of the high-gain Lyapunov function for this network.

ii. Assuming a large gain, what are the stable equilibrium points for this Hopfield network? What can you say about the basins of attraction for these stable equilibrium points? Explain your answers.

E21.10 Repeat E21.9 for the following weight and bias:

$$\mathbf{W} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$



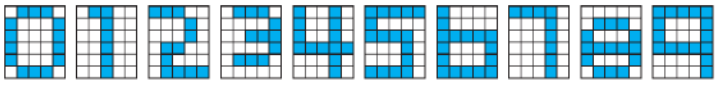
E21.11 In Exercise E7.11 we asked the question: How many prototype patterns can be stored in one weight matrix? Repeat that problem using the Hopfield network. Begin with the digits "0" and "1". (The digits are shown at the end of this problem.) Add one digit at a time up to "6", and test how often the correct digit is reconstructed after randomly changing 2, 4 and 6 pixels.

i. First use the Hebb rule to create the weight matrix for the digits "0"

and "1". Then randomly change 2 pixels of each digit and apply the noisy digits to the network. Repeat this process 10 times, and record the percentage of times in which the correct pattern (without noise) is produced at the output of the network. Repeat as 4 and 6 pixels of each digit are modified. The entire process is then repeated when the digits "0", "1" and "2" are used. This continues, one digit at a time, until you test the network when all of the digits "0" through "6" are used. When you have completed all of the tests, you will be able to plot three curves showing percentage error versus number of digits stored, one curve each for 2, 4 and 6 pixel errors.

ii. Repeat part (i) using the pseudoinverse rule (see Chapter 7), and compare the results of the two rules.

iii. For extra credit, repeat part (i) using the method described in [LiMi89]. In that paper it is called Synthesis Procedure 5.1.



$\mathbf{p}_1$ $\mathbf{p}_2$ $\mathbf{p}_3$ $\mathbf{p}_4$ $\mathbf{p}_5$ $\mathbf{p}_6$ $\mathbf{p}_7$ $\mathbf{p}_8$ $\mathbf{p}_9$ $\mathbf{p}_{10}$

22 Practical Training Issues

Objectives	22-1
Theory and Examples	22-2
Pre-Training Steps	22-3
Selection of Data	22-3
Data Preprocessing	22-5
Choice of Network Architecture	22-8
Training the Network	22-13
Weight Initialization	22-13
Choice of Training Algorithm	22-14
Stopping Criteria	22-14
Choice of Performance Function	22-16
Multiple Training Runs and Committees of Networks	22-17
Post-Training Analysis	22-18
Fitting	22-18
Pattern Recognition	22-21
Clustering	22-23
Prediction	22-24
Overfitting and Extrapolation	22-27
Sensitivity Analysis	22-28
Epilogue	22-30
Further Reading	22-31

Objectives

Previous chapters have focused on particular neural network architectures and training rules, with an emphasis on fundamental understanding. In this chapter, we will discuss some practical training tips that apply to a variety of networks. No derivations are provided for the techniques that are presented here, but we have found these methods to be useful in practice.

There will be three basic sections in this chapter. The first section describes things that need to be done prior to training a network, such as collecting and preprocessing data and selecting the network architecture. The second section addresses network training itself. The final section considers post-training analysis.

Theory and Examples

In previous chapters, we have discussed a variety of neural network architectures and learning rules. Those chapters have placed special emphasis on the fundamental concepts behind each network. In this chapter, we will concentrate on practical aspects of training neural networks. Theoretical aspects and practical aspects are not mutually exclusive. It is only by combining a deep knowledge of network fundamentals with practical experience in using neural networks that we can get the most out of this technology.

Figure 22.1 illustrates the neural network training process. It is an iterative procedure that begins by collecting data and preprocessing it to make training more efficient. At this stage, the data also needs to be divided into training/validation/testing sets (see Chapter 13). After the data is selected, we need to choose the appropriate network type (multilayer, competitive, dynamic, etc.) and architecture (e.g., number of layers, number of neurons). Then we select a training algorithm that is appropriate for the network and the problem we are trying to solve. After the network is trained, we want to analyze the performance of the network. This analysis may lead us to discover problems with the data, the network architecture, or the training algorithm. The entire process is then iterated until the network performance is satisfactory.

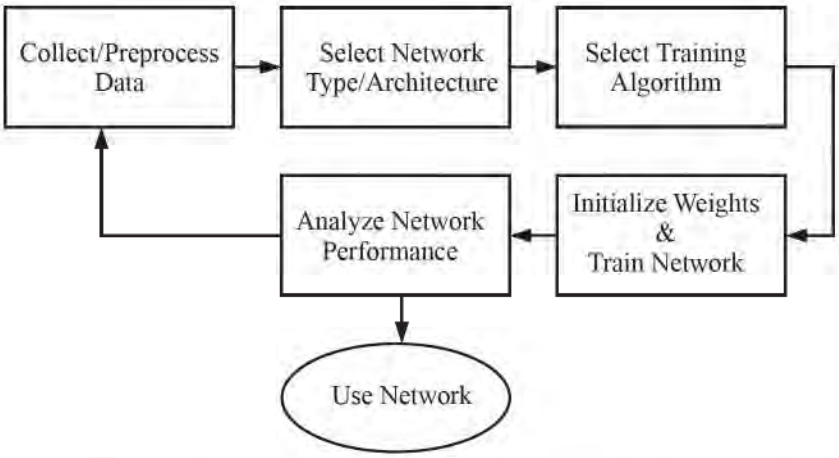


Figure 22.1 Flowchart of Neural Network Training Process

In the remainder of this chapter, we will discuss each part of the training process in some detail. We have divided this material into three major sections: Pre-Training Steps, Training the Network, and Post-Training Analysis.

Before we dig into the details of training, it is worth making a preliminary comment. Before beginning the neural network training process, you should first determine if a neural network is needed to solve your problem, or if some simpler linear technique might be adequate. For example, there is no need to use a neural network for a fitting problem, if standard linear regression will produce a satisfactory result. The neural network techniques provide additional power, but at the expense of more challenging training requirements. When linear methods will work, they are the first choice.

Pre-Training Steps

There are a number of steps that need to be performed before the network is trained. They can be grouped into three categories: Selection of Data, Data Preprocessing, and Choice of Network Type and Architecture.

Selection of Data

It is generally difficult to incorporate prior knowledge into a neural network, therefore the network will only be as good as the data that is used to train it. Neural networks represent a technology that is at the mercy of the data. The training data must span the full range of the input space for which the network will be used. As we discussed in Chapter 13, there are training methods we can use to insure that the network interpolates accurately throughout the range of the data provided (generalizes well). However, it is not possible to guarantee network performance when the inputs to the network are outside the range of the training set. Neural networks, like other nonlinear "black box" methods, do not extrapolate well.

It is not always easy to be sure that the input space is adequately sampled by the training data. For simple problems, in which the dimension of the input vector is small, and each element of the input vector can be chosen independently, we can sample the input space using a grid. However, these conditions are not often satisfied. For many problems, the dimension of the input space is large, which precludes the use of grid sampling. In addition, it is often the case that the input variables are dependent. For example, consider Figure 22.2. The shaded area represents the range over which the two inputs can vary. Even though each variable can range from -1 to 1, we would not need to create a grid in which both variables vary throughout their range, as shown by the dots in Figure 22.2. The network only needs to fit the function in the shaded area, since this is where the network will be used. It would be inefficient to fit the network outside the range of its use. This is especially true when the input dimension is large.

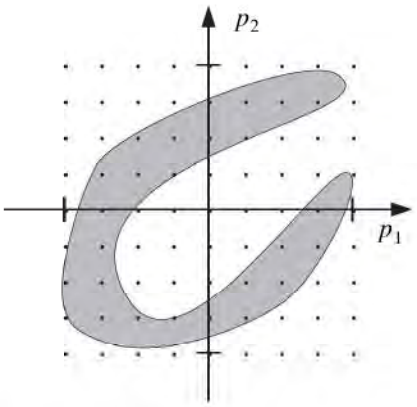


Figure 22.2 Input Range With Dependent Input Variables

It may not be possible to precisely define the active region of the input space. However, we can often collect data during standard operation of the system we are trying to model. In some cases, we have complete control over the design of the experiment during which data is collected. In these cases, we must be sure that the experimental setup drives the system through all conditions for which we plan to use the network.

How can we be sure that the input space has been adequately sampled by the training data? This is difficult to do prior to training, and there are many cases in which we have no control over the data collection process and must use whatever data is available. We will come back to this subject in the Post-Training Analysis section on page 22-18. By analyzing the trained network, we can often tell if the training data was sufficient. In addition, we can use techniques that indicate when a network is being used outside the range of the data with which it was trained. This will not improve the network performance, but it will prevent us from using a network in situations where it is not reliable.

After collecting the data, we generally divide it into three sets: training, validation and testing. As we discussed in Chapter 13, the training set will generally make up approximately 70% of the full data set, with validation and testing making up approximately 15% each. It is important that each of these sets be representative of the full data set — that the validation and test sets cover the same region of the input space as the training set. The simplest method for dividing the data is to select each set at random from the full data set. This usually produces a good result, but it is best to review the division to check for major differences between the sets. It is also possible in the post-training analysis to detect problems in the data division. We will have more to say about that later.

A final question we must ask about the selection of data is "Do we have enough data?" This is difficult to answer, especially before we train the net-

work. The amount of data that is required depends on the complexity of the underlying function that we are trying to approximate (or the complexity of the decision boundaries that we are trying to implement). If the function to be approximated is very complex, with many inflection points, then this requires a large amount of data. If the function is very smooth, then the data requirements are significantly reduced (unless the data is very noisy). The choice of the data set size is closely related to the choice of the number of neurons in the neural network. This is discussed in the Choice of Network Architecture section on page 22-8. Of course, we generally don't know how complex the underlying function is before we begin training the network. For this reason, as we discussed earlier, the entire neural network training process is iterative. At the completion of training we will analyze the performance of the network. The results of that analysis can help us decide if we have enough data or not.

Data Preprocessing

The main purpose of the data preprocessing stage is to facilitate network training. Data preprocessing consists of such steps as normalization, nonlinear transformations, feature extraction, coding of discrete inputs/targets, handling of missing data, etc. The idea is to perform preliminary processing of the data to make it easier for the neural network training to extract the relevant information.

For example, in multilayer networks, sigmoid transfer functions are often used in the hidden layers. These functions become essentially saturated when the net input is greater than three ($\exp(-3) \cong 0.05$). We don't want this to happen at the beginning of the training process, because the gradient will then be very small. In the first layer, the net input is a product of the input times the weight plus the bias. If the input is very large, then the weight must be small in order to prevent the transfer function from becoming saturated. It is standard practice to normalize the inputs before applying them to the network. In this way, initializing the network weights to small random values guarantees that the weight-input product will be small. Also, when the input values are normalized, the magnitudes of the weights have a consistent meaning. This is especially important when using regularization, as described in Chapter 13. Regularization requires the weight values to be small. However, "small" is a relative term; if the input values are very small, we need large weights to produce a significant net input. Normalizing the inputs clarifies the meaning of "small" weights.

Normalization

There are two standard methods for *normalization*. The first method normalizes the data so that they fall into a standard range — typically -1 to 1. This can be done with

$$\mathbf{p}^n = 2(\mathbf{p} - \mathbf{p}^{min})./(\mathbf{p}^{max} - \mathbf{p}^{min}) - 1, \qquad (22.1)$$

where $\mathbf{p}^{min}$ is the vector containing the minimum values of each element of the input vectors in the data set, $\mathbf{p}^{max}$ contains the maximum values, ./

represents an element-by-element division of two vectors, and $\mathbf{p}^n$ is the resulting normalized input vector.

An alternative normalization procedure is to adjust the data so that they have a specified mean and variance — typically 0 and 1. This can be done with the transformation

$$\mathbf{p}^n = (\mathbf{p} - \mathbf{p}^{mean})./\mathbf{p}^{std}, \tag{22.2}$$

where $\mathbf{p}^{mean}$ is the average of the input vectors in the data set, and $\mathbf{p}^{std}$ is the vector containing the standard deviations of each element of the input vectors.

Generally, the normalization step is applied to both the input vectors and the target vectors in the data set.

Nonlinear Transformation

In addition to normalization, which involves a linear transformation, *nonlinear transformations* are sometimes also performed as part of the preprocessing stage. Unlike normalization, which is a standard process that can be applied to any data set, these nonlinear transformations are case-specific. For example, many economic variables show a logarithmic dependence [BoJe94]. In that case, it might be appropriate to take the logarithm of the input values before applying them to the neural network. Another example is molecular dynamics simulation [RaMa05], in which atomic forces are calculated as functions of distances between atoms. Since it is known that the forces are inversely related to the distances, we might perform the reciprocal transformation on the inputs, before applying them to the network. This represents one way of incorporating prior knowledge into neural network training. If the nonlinear transformation is cleverly chosen, it can make the network training more efficient. The preprocessing will off-load some of the work required of the neural network in finding the underlying transformation between inputs and targets.

Feature Extraction

Another data preprocessing step is called *feature extraction*. This generally applies to situations in which the dimension of the raw input vectors is very large and the components of the input vector are redundant. The idea of feature extraction is to reduce the dimension of the input space by calculating a small set of features from each input vector, and using the features as the input to the neural network. For example, neural networks can be used to analyze EKG (electrocardiogram) signals to identify heart problems [HeOh97]. The EKG might involve 12 or 15 signals (leads) measured over several minutes at a high sampling rate. This is too much data to apply directly to the neural network. Instead, we would extract certain features from the EKG signal, such as average time intervals between certain waveforms, average amplitudes of certain waves, etc. (See Chapter 25.)

Principal Components

There are also certain general-purpose feature extraction methods. One of these is the method of *principal component analysis* (PCA) [Joll02]. This method transforms the original input vectors so that the components of the

transformed vectors are uncorrelated. In addition, the components of the transformed vector are ordered such that the first component has the greatest variance, the second component has the next greatest variance, etc. We generally keep only the first few components of the transformed vector, which account for most of the variance in the original vector. This results in a large reduction in the dimension of the input vector, if the original components are highly correlated. The drawback of using PCA is that it only considers linear relationships between the components of the input vector. When reducing the dimension using a linear transformation, we might lose some nonlinear information. Since the main purpose of using neural networks is to gain the power of their nonlinear mapping capabilities, we should be careful when using principal components to reduce the input dimension before applying the inputs to the neural network. There is a nonlinear version of PCA, called kernel PCA [ScSm99].

Coding the Targets

Another important preprocessing step is needed whenever the inputs or targets take on only discrete values. For example, in pattern recognition problems, each target will represent one of a finite number of classes. In these cases we need to have a procedure for *coding the inputs or targets*. If we have a pattern recognition problem in which there are four classes, there are at least three common ways in which we could code the targets. First, we can have scalar targets that take on four possible values (e.g., 1, 2, 3, 4). Second, we can have two-dimensional targets, which represent a binary code of the four classes (e.g., (0,0), (0,1), (1,0), (1,1)). Third, we can have four-dimensional targets, in which only one neuron at a time is active (e.g., (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)). The third method tends to yield the best results in our experience. (Note that discrete inputs can be coded in the same ways as discrete targets.)

When coding the target values, we also need to consider the transfer function that is used in the output layer of the network. For pattern recognition problems, we would typically use sigmoid functions: log-sigmoid or tangent-sigmoid. If we use the tangent-sigmoid in the last layer, which is more common, then we might consider assigning target values to -1 or 1, which represent the asymptotes of the function. However, this tends to cause difficulties for the training algorithm, which tries to saturate the sigmoid function to meet the target value. It is better to assign target values at the point where the second derivative of the sigmoid function is maximum (see [LeCu98]). For the tangent-sigmoid function, this occurs when the net input is -1 and 1, which corresponds to output values of -0.76 and +0.76.

Softmax

Another transfer function that is used in the output layer of multilayer pattern recognition networks is the *softmax* function. This transfer function has the form

$$a_i = f(n_i) = \exp(n_i) \div \sum_{j=1}^{S} \exp(n_j) . \tag{22.3}$$

The outputs of the softmax transfer function can be interpreted as the probabilities associated with each class. Each output will fall between 0 and 1, and the sum of the outputs will equal 1. See Chapter 24 for an example application of the softmax transfer function.

Missing Data

Another practical issue to consider is *missing data*. It is often the case, especially when dealing with economic data, for example, that some of the data is missing. For instance, we might have an input vector containing 20 economic variables that are collected at monthly intervals. There may be some months in which one or two of the 20 variables were not collected properly. The simplest solution to this problem would be to throw out the data for any month in which any of the variables were missing. However, we might be very limited in the amount of data available, and it might be very expensive to collect additional data. In these cases, we would like to make full use of any data that we have, even if it is incomplete.

There are several strategies for dealing with missing data. If the missing data occurs in an input variable, one possibility is to replace the missing value with the average value for that particular input variable. At the same time, we could add an additional flag element to the input vector that would indicate that missing data for that input variable had been replaced with the average. This additional element of the input vector could be assigned the value 1 when the input variable was available, and 0 when the input variable was missing for that case. This would provide the neural network with information about which variables were missing. An additional flag element would be added to the input vector for every input variable that contained missing points.

If the missing data occurs in an element of the target, then the performance index can be modified so that errors associated with the missing target values are not included. All known target values will contribute to the performance index, but missing target values will not.

Choice of Network Architecture

The next step in the network training process is the choice of network architecture. The basic type of network architecture is determined by the type of problem we wish to solve. Once the basic architecture is chosen, we need to decide such specific details as how many neurons and layers we want to use, how many outputs the network should have, and what type of performance function we want to use for training.

Choice of Basic Architecture

The first step in choosing the architecture is to define the problem that we are trying to solve. For this chapter, we will limit our discussion to four types of problems: fitting, pattern recognition, clustering and prediction.

Fitting

Fitting is also referred to as function approximation or regression. In fitting problems, you want a neural network to map between a set of inputs and a

corresponding set of targets. For example, a realtor might want to estimate home prices from such input variables as tax rate, pupil/teacher ratio in local schools and crime rate. An automotive engineer might want to estimate engine emission levels based on measurements of fuel consumption and speed. A physician might want to predict a patient's body fat level based on body measurements. For fitting problems, the target variable takes on continuous values. (For an example of training a neural network for a fitting problem, see Chapter 23.)

The standard neural network architecture for fitting problems is the multilayer perceptron, with tansig neurons in the hidden layers, and linear neurons in the output layer. The tansig transfer function is generally preferred over the logsig transfer function in the hidden layers for the same reason that inputs are normalized. It produces outputs (which are inputs to the next layer) that are centered near zero, whereas the logsig transfer function always produces positive outputs. For most fitting problems, a single hidden layer is sufficient. If the results with one hidden layer are not satisfactory, two layers are sometimes used. It would be rare in a standard fitting problem to use more than two hidden layers, although, for very difficult problems, deep networks, with many layers, have been used. Linear transfer functions are used in the output layer for fitting problems, because the target output is a continuous variable. As we saw in Chapter 11, a two layer network with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer is a universal approximator.

Radial basis networks can also be used for fitting problems. The Gaussian transfer function is most commonly used in the hidden layer for these networks, with linear transfer functions in the output layer.

Pattern Recognition

Pattern recognition is also referred to as pattern classification. In pattern recognition problems, you want a neural network to classify inputs into a set of target categories. For example, a wine dealer might want to recognize the vineyard that a particular bottle of wine came from, based on a chemical analysis of the wine. A physician might want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness and mitosis.

In addition to fitting problems, multilayer perceptrons can be used for pattern recognition. The main difference between a fitting network and a pattern recognition network is the transfer function used in the output layer. For pattern recognition problems, we generally use a sigmoid function in the output layer. The radial basis function network can also be used for pattern recognition.

For an example of training a neural network for a pattern recognition problem, see Chapter 25.

Clustering

Clustering, or segmentation, is another use for neural networks. In clustering problems, you want a neural network to group data by similarity. For example, businesses may wish to perform market segmentation, which is

done by grouping people according to their buying patterns. Computer scientists may want to perform data mining by partitioning data into related subsets. Biologists may wish to perform bioinformatic analyses, such as grouping genes with related expression patterns.

Any of the competitive networks described in Chapter 16 could be used for clustering. The self-organizing feature map (SOFM) is the most popular network for clustering. The main advantage of the SOFM is that it allows visualization of high-dimensional spaces.

For an example of training a neural network for a clustering problem, see Chapter 26.

Prediction

Prediction also falls under the categories of time series analysis, system identification, filtering or dynamic modeling. The idea is that we wish to predict the future value of some time series. An equities trader might want to predict the future value of some security. A control engineer might want to predict a future value of the concentration of some chemical, which is the output of a processing plant. A power systems engineer might want to predict outages on the electric grid.

Prediction requires the use of dynamic neural networks, as discussed in Chapter 14. The specific form of the network will depend on the particular application. The simplest network for nonlinear prediction is the focused time-delay neural network, which is shown in Figure 22.3. This is part of a general class of dynamic networks, called focused networks, in which the dynamics appear only at the input layer of a static multilayer feedforward network. This network has the advantage that it can be trained using static backpropagation algorithms, since the tapped-delay-line at the input of the network can be replaced with an extended vector of delayed values of the input.

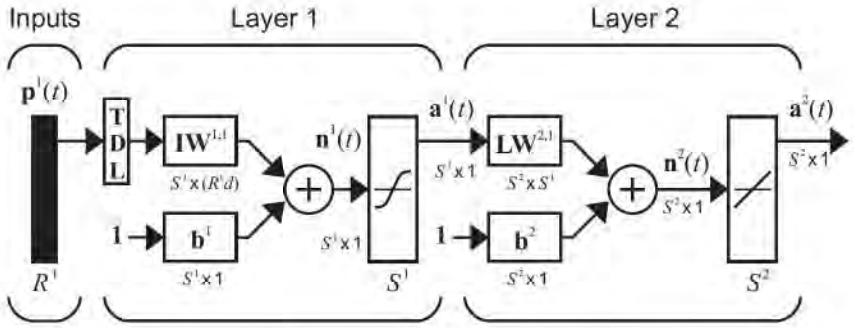


Figure 22.3 Focused Time-Delay Neural Network

For problems of dynamic modeling and control, the NARX network (Nonlinear AutoRegressive model with eXogenous input) is popular. This network is shown in Figure 22.4. The input signal could represent, for

example, the voltage applied to a motor, and the output could represent the angular position of a robot arm. As with the focused time-delay neural network, the NARX network can be trained with static backpropagation. The two tapped-delay-lines can be replaced with extended vectors of delayed inputs and targets. We can use targets, instead of feeding back the network outputs (which would require dynamic backpropagation for training), because the output of the network should match the targets when training is complete.This is discussed in more detail in Chapter 27.

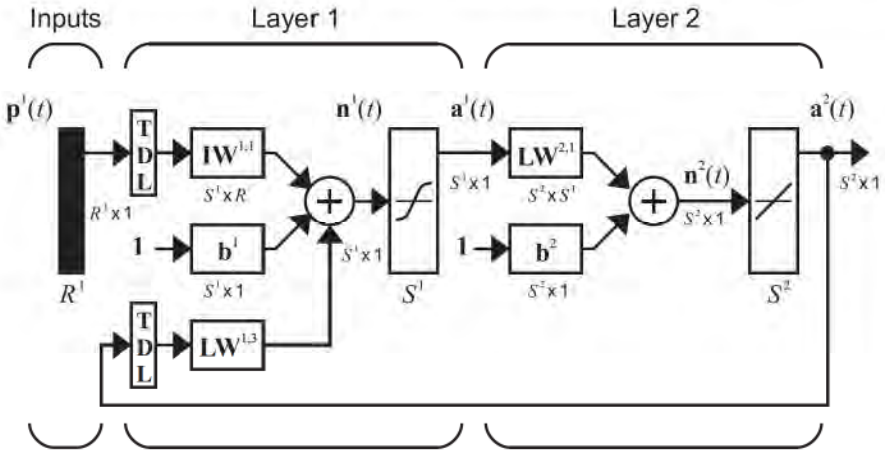


Figure 22.4 NARX Neural Network

There are many other types of dynamic networks that could be used for prediction, but the focused time delay network and the NARX network are the simplest of their type.

For an example of training a neural network for a prediction problem, see Chapter 27.

Selection of Architecture Specifics

After the basic network structure is chosen, we want to select the specifics of the architecture (e.g., the number of layers, the number of neurons, etc.). In some cases, the basic architecture choice will automatically determine the number of layers. For example, if the SOFM is used for clustering, then the network will have one layer. In the case of the multilayer network, which can be used for fitting or pattern recognition, the number of hidden layers is not determined by the problem, since any number of hidden layers is possible. The standard procedure is to begin with a network with one hidden layer. If the performance of the two-layer network is not satisfactory, then a three-layer network can be used. It would be unusual to use more than two hidden layers. The training becomes more difficult when multiple hidden layers are used. This is because each layer performs a squashing operation, as sigmoid functions are used in the hidden layers. This causes the

derivatives of the performance function with respect to weights in the early layers to be quite small, which can cause slow convergence for steepest descent optimization. For very difficult problems, however, deep networks, with several hidden layers, can be used. Typically, parallel or GPU computing is required to train deep multilayer networks within a reasonable amount of time.

We also need to select the number of neurons in each layer. The number of neurons in the output layer is the same as the size of the target vector. The numbers of neurons in the hidden layers are determined by the complexities of the function that is being approximated or the decision boundaries that are being implemented. Unfortunately, we don't normally know how complex the problem is until we try to train the network. The standard procedure is to begin with more neurons than necessary and then to use early stopping or Bayesian regularization to prevent overfitting, as was described in Chapter 13.

The principal drawback to having too many neurons is that the network may overfit the data. If we use early stopping or Bayesian regularization, then we can prevent overfitting. However, there may be some situations in which we are concerned with the computation time or space required by the network (e.g., for real-time implementation on microcontrollers, VLSI or FPGAs). In these cases we want to find the simplest network that will fit the data. If you use Bayesian regularization, the effective number of parameters can be used to determine how many neurons to use. If, after training, the effective number of parameters is much less than the total number of parameters in the network, then the number of neurons can be reduced, and the network retrained. It is also possible to use "pruning" methods to eliminate neurons or weights in the network.

The number of neurons in the last layer is equal to the number of elements in the target vector. However, when there are multiple targets, we have a choice to make. We can have one network with multiple outputs, or we can have multiple networks, each with one output. For example, neural networks have been used to estimate LDL, VLDL and HDL cholesterol levels, based on a spectral analysis of the blood. It is possible to have one neural network with three neurons in the output layer to estimate all three cholesterol levels, or we could have three neural networks, with each one estimating only one of the three components. Theoretically, both methods should work, but in practice one method may work better than another. We generally start with one multi-output network, and then use multiple single-output networks if the original results are not satisfactory.

Another architectural choice is the size of the input vector. This is often a simple choice, which is determined by the training data. However, there are times when input vectors in the training data have redundant or irrelevant elements. When the dimension of the potential input vector is very large, it is sometimes advantageous to eliminate redundant or irrelevant elements. This can reduce the required computation and can assist in pre-

Input Selection

venting overfitting during training. The *input selection* process for nonlinear networks can be quite difficult, and there is no perfect solution. The Bayesian regularization method (Eq. (13.23)) can be modified to assist in input selection. It is possible to have different α parameters for different sets of weights. For example, we can let each column of the weight matrix in the first layer of a multilayer network have its own α parameter. If a given element of the input vector is irrelevant, then the corresponding α parameter would become large and force all elements of that column of the weight matrix to be small. That element could then be eliminated from the input vector.

Another technique that can assist in pruning the input vector is a sensitivity analysis of the trained network. In the Sensitivity Analysis section on page 22-28 we discuss this technique.

Training the Network

After the data has been prepared, and the network architecture has been selected, we are ready to train the network. In this section, we will discuss some of the decisions that need to be made as part of the training process. This includes the method for initializing the weights, the training algorithm, the performance index, and the criterion for stopping training.

Weight Initialization

Before training the network, we need to initialize the weights and biases. The method we use will depend on the type of network. For multilayer networks, the weights and biases are generally set to small random values (e.g., uniformly distributed between -0.5 and 0.5, if the inputs are normalized to fall between -1 and 1). As we discussed in Chapter 12, if we set the weights and biases to zero, the initial condition may fall on a saddle point of the performance surface. If we make the initial weights large, the initial condition can fall on a flat part of the performance surface, caused by saturation of the sigmoid transfer functions.

There is another approach to setting the initial weights and biases for a two-layer network. It was introduced by Widrow and Nguyen [WiNg90]. The idea is to set the magnitude of the weights in the first layer so that the linear region of each sigmoid function covers $1/S^1$ of the range of the input. The biases are then randomly set, so that the center of each sigmoid function falls randomly in the input space. The details of the method are as follows (assuming the inputs to the network have been normalized to values between -1 and 1).

Set row i of $\mathbf{W}^1$, ${}_i\mathbf{w}^1$, to have a random direction and a magnitude of

$$\left\| {}_i\mathbf{w}^1 \right\| = 0.7(S^1)^{1/R}.$$

Set b_i to a uniform random value between $-\|_i \mathbf{w}^1\|$ and $\|_i \mathbf{w}^1\|$.

For competitive networks, the weights can also be set as small random numbers. Another possibility is to randomly select some of the input vectors in the training set to become initial rows of the weight matrix. In this way, we can be sure that the initial weights will fall within the range of the input vectors, so we will be less likely to have dead units, as described in Chapter 16. For the SOM, dead units are not a problem. The initial neighborhood size is set large enough so that all neurons will have the opportunity to learn during the initial stages of training. This will move all weight vectors into the appropriate region of the input space. Training can converge faster, however, if rows of the weight matrix are initially placed in the active input region.

Choice of Training Algorithm

For multilayer networks, we generally use gradient- or Jacobian-based algorithms, as described in Chapter 12. These algorithms can be implemented in either batch mode or sequential (also known as incremental, pattern or stochastic) mode. For example, in the sequential form of steepest descent (see Eq. (11.13)) we update the weights after each input is presented to the network. In batch mode (see page 12-7), all of the inputs are presented to the network, and the total gradient is computed by summing the gradients for each input, before the weights are updated. In some situations, the sequential form is preferred — for example, when on-line or adaptive operation is required. However, many of the more efficient optimization algorithms (e.g., conjugate gradient and Newton's methods) are inherently batch algorithms.

For multilayer networks with up to a few hundred weights and biases that are being used for function approximation, the Levenberg-Marquardt algorithm (see Eq. (12.31)) is usually the fastest training method. When the number of weights reaches a thousand or more, the Levenberg-Marquardt algorithm is not as efficient as some of the conjugate gradient algorithms. This is mainly because the matrix inverse calculation scales geometrically with the number of weights. For large networks, the Scaled Conjugate Gradient algorithm of [Moll93] is very efficient. This method is also attractive for pattern recognition problems. The Levenberg-Marquardt algorithm does not work as well for pattern recognition, in which the sigmoid transfer functions in the final layer are operating well outside the linear region.

Of the algorithms that can be implemented in sequential mode, the fastest are the extended Kalman filter algorithms. These algorithms are closely related to sequential implementations of the Gauss-Newton algorithm. Unlike the batch version of Gauss-Newton, they do not require an inversion of the approximate Hessian matrix. The decoupled extended Kalman filter implementation of [PuFe97] appears to be the most efficient of these types of algorithms.

Stopping Criteria

For most applications of neural networks, the training error never converges identically to zero. The error can reach zero for the perceptron network, assuming a linearly separable problem, as we showed in Chapter 4. However, it is unlikely to happen for multilayer networks. For this reason, we need to have other criteria for deciding when to stop the training.

We can stop the training when the error reaches some specified limit. However, it is usually difficult to know what an acceptable error level is. The simplest criterion is to stop the training after a fixed number of iterations. Because it is also difficult to know how many iterations will be required, the maximum iteration number is generally set reasonably high. If the weights have not converged after the maximum number of iterations has been reached, we can restart training, using the final weights from the first run as initial conditions for the restart. (We will talk more about how to tell if a network has converged in the Post-Training Analysis section on page 22-18.)

Another stopping criterion is the norm of the gradient of the performance index. If this norm reaches a sufficiently small threshold, then the training can be stopped. Since the gradient should be zero at a minimum of the performance index, this criterion will stop the algorithm when it gets close to the minimum. Unfortunately, as we have seen in Chapter 12, the performance surface for multilayer networks can have many flat regions, where the norm of the gradient will be small. For this reason, the threshold for the minimum norm should be set to a very small value (e.g., 10^{-6} for mean square error performance indices, with normalized targets), so that the training does not end prematurely.

We can also stop the training when the reduction in the performance index per iteration becomes small. As with the norm of the gradient, this criterion can stop the training too early. During the training of multilayer networks, the performance can remain almost constant for a number of iterations before dropping suddenly. When training is complete, it is useful to view the training performance curve on a log-log scale, as in Figure 22.5, to verify convergence.

If we are using early stopping, as discussed in Chapter 13, to prevent overfitting, then we will stop the training when the performance on the validation set increases for a set number of iterations. In addition to preventing overfitting, this stopping procedure also provides a significant reduction in computation; for most practical problems, the validation error will increase before any of the other stopping criteria are reached.

As shown in Figure 22.1, neural network training is an iterative process. Even after the training algorithm has converged, post-training analysis may suggest that the network be modified and retrained. In addition, several training runs should be made for each potential network to ensure that a global minimum has been reached.

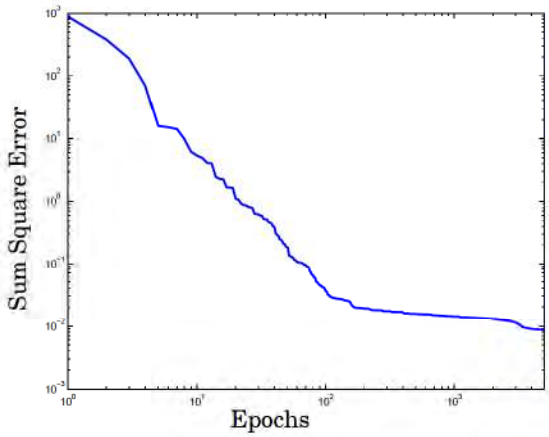


Figure 22.5 Typical Training Performance Curve

The previous stopping criteria apply mainly to gradient-based training. When training competitive networks, like the SOFM, there is no explicit performance index or gradient to monitor for convergence. The training stops only when the maximum number of iterations has been reached. For SOFMs, the learning rate and the neighborhood size are decreased over time. Typically, the neighborhood size is reduced to zero by the completion of training, so the maximum iteration number determines the end of training, as well as the rate of decrease in neighborhood size and learning rate. This is therefore a very important parameter. It is generally chosen to be more than ten times the number of neurons in the network. This is an approximate number, and the network needs to be analyzed at the completion of training to determine if the performance is satisfactory. (This will be discussed in the Post-Training Analysis section on page 22-18.) The network may need to be trained several times with different training lengths to achieve a satisfactory result.

Choice of Performance Function

For multilayer networks, the standard performance index is mean square error. When all inputs in the training set are equally likely to occur, then this can be written

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q), \tag{22.4}$$

or

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}(t_{i,q}-a_{i,q})^2. \tag{22.5}$$

The scale factor that occurs outside the sum has no effect on the location of the optimum weights. Therefore, the sum square error performance index will produce the same weights as the mean square error performance index. However, the appropriate scaling can be useful when comparing errors on data sets of different size.

While mean square error is the most common performance index, there are others that have been used. For example, we could use mean absolute error. This would be similar to Eq. (22.5), except that the absolute value of the error would be used, instead of the square of the error. This performance index is generally less sensitive to one or two large errors in the data set, and is therefore somewhat more robust to outliers than is the mean square error algorithm. This concept can be extended to any power of the absolute error, as follows

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}\left|t_{i,q}-a_{i,q}\right|^K, \tag{22.6}$$

where $K = 2$ corresponds to mean square error and $K = 1$ corresponds to mean absolute error. The general error given by Eq. (22.6) is referred to as the Minkowski error.

As we saw in Chapter 13, the mean square performance index can be augmented with the mean square weights, to produce a regularized performance index, which is used to prevent overfitting. The Bayesian regularization algorithm is an excellent training method for preventing overfitting. It uses a regularized performance index, and uses Bayesian methods to select the regularization parameter. See Chapter 13 for details.

Mean square error works well for function approximation problems, in which the target values are continuous. However, in pattern recognition problems, where the targets take on discrete values, other performance indices might be more appropriate. One performance index that has been proposed for classification problems is *cross-entropy* [Bish95]. Cross-entropy is defined as

Cross-Entropy

$$F(\mathbf{x}) = -\sum_{q=1}^{Q}\sum_{i=1}^{S^M} t_{i,q}\ln\frac{a_{i,q}}{t_{i,q}}. \tag{22.7}$$

Here we assume that the target values are 0 and 1, and they identify which of the two classes the input vector belongs to. The softmax transfer function

is generally used in the last layer of the neural network, if the cross-entropy performance index is used.

As a closing note on the choice of performance index, recall from Chapter 11 that the backpropagation algorithm for computing training gradients will work for any differentiable performance index. If you change the performance index, you only need to change the initialization of the sensitivities in the last layer (see Eq. (11.37)).

Multiple Training Runs and Committees of Networks

A single training run may not produce optimal performance, because of the possibility of reaching a local minimum of the performance surface. It is best to restart the training at several different initial conditions and select the network that produces the best performance. Five to ten restarts will almost always produce a global optimum [HaBo07].

There is another way to perform multiple training runs and make use of all of the networks that have been trained. This is called the committee of networks [PeCo93]. For each training session, the validation set is randomly selected from the training data, and a random set of initial weights and biases is chosen. After N networks have been trained, all of the networks are used together to form a joint output. For function approximation networks, the joint output can be a simple average of the outputs of each network. For classification networks, the joint output can be the result of a vote, in which the class that is chosen by the majority of the networks is selected as the output of the committee. The performance of the committee will usually be better than even the best of the individual networks. In addition, the variation in the outputs of the individual networks can be used to provide error bars, or confidence levels, for the committee output.

Post-Training Analysis

Before using a trained neural network, we need to analyze it to determine if the training was successful. There are many techniques for post-training analysis. We will discuss some of the more common ones. Since these techniques vary, depending on the application, we will organize them according to these four application areas: fitting, pattern recognition, clustering and prediction.

Fitting

One useful tool for analyzing neural networks trained for fitting problems is a regression between the trained network outputs and the corresponding targets. We fit a linear function of the form

$$a_q = mt_q + c + \varepsilon_q, \tag{22.8}$$

where m and c are the slope and offset, respectively, of the linear function, t_q is a target value, a_q is a trained network output, and ε_q is the residual error of the regression.

The terms in the regression can be computed as follows:

$$\hat{m} = \frac{\sum_{q=1}^{Q} (t_q - \dot{t})(a_q - \bar{a})}{\sum_{q=1}^{Q} (t_q - \dot{t})^2}, \tag{22.9}$$

$$\hat{c} = \bar{a} - \hat{m}\dot{t}, \tag{22.10}$$

where

$$\bar{a} = \frac{1}{Q}\sum_{q=1}^{Q} a_q, \; \dot{t} = \frac{1}{Q}\sum_{q=1}^{Q} t_q. \tag{22.11}$$

Figure 22.6 shows an example regression analysis. The blue line represents the linear regression, the thin black line represents the perfect match $a_q = t_q$, and the circles represent the data points. In this example, we can see that the match is pretty good, although not perfect. The next step would be to investigate data points that fall far from the regression line. For example, there are two points around $t = 27$ and $a = 17$ that seem to be *outliers*. We would investigate these points to see if there was a problem with the data. This could be a bad data point, or it could be located far from other training points. In the latter case, we would need to collect more data in that region.

Outliers

In addition to computing the regression coefficients, we often also compute the correlation coefficient between the t_q and a_q, which is also known as the R value:

$$R = \frac{\sum_{q=1}^{Q} (t_q - \dot{t})(a_q - \bar{a})}{(Q-1)s_t s_a}, \tag{22.12}$$

where

$$s_t = \sqrt{\frac{1}{Q-1}\sum_{q=1}^{Q} (t_q - \dot{t})} \text{ and } s_a = \sqrt{\frac{1}{Q-1}\sum_{q=1}^{Q} (a_q - \bar{a})}. \tag{22.13}$$

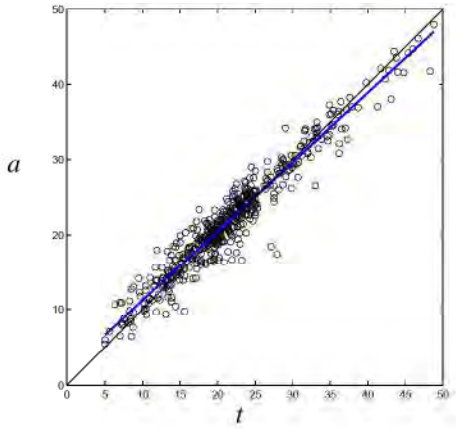


Figure 22.6 Regression Between Trained Network Outputs and Targets

The R value can generally range from -1 to 1, but we would expect it to be close to 1 for our neural network application. If $R = 1$, then all of the data points will fall exactly on the regression line. If $R = 0$, then the data will not be concentrated around the regression line, but will be randomly scattered. For the data of Figure 22.6, $R = 0.965$. We can see that the data does not fall exactly on the regression line, but the variation is relatively small.

The square of the correlation coefficient, R^2, is sometimes used instead of R. R^2 represents the proportion of the variability in a data set that is accounted for by the linear regression, and is also referred to as the coefficient of determination. For the data of Figure 22.6, $R^2 = 0.931$.

When the R or R^2 values are significantly less than 1, then the neural network has not done a good job of fitting the underlying function. A close analysis of the scatter plot may be helpful in determining problems in the fit. For example, we might find that when the targets are large there is more spread in the scatter plot. (This is not the case in Figure 22.6.) We might also notice that there are fewer data points with large targets. This would indicate that we need to have more data points in the training set for these target values.

Recall that the original data set was divided into training, validation (if early stopping is used) and testing subsets. The regression analysis should be performed on each subset individually, as well as the full data set. Differences between the subsets would indicate overfitting or extrapolation. For example, if the training set shows accurate fitting, but the validation and test results are poor, then this would indicate overfitting (which can sometimes happen, even when early-stopping is used). In this case, we might reduce the size of the neural network and retrain. If both the training and validation results are good, but the testing results are poor, then

this could indicate extrapolation (where the testing data falls outside the training and validation data). In this case, we need to provide more data for training and validation. If the results for all three data sets are poor, then it might be necessary to increase the number of neurons in the network. Another choice is to increase the number of layers in the network. If you start with a single hidden layer, and the results are poor, then a second hidden layer could be helpful. First, try more neurons in the single hidden layer, and then increase the number of layers.

In addition to the regression/scatter plot, another tool that can identify outliers is a histogram of the errors, as shown in Figure 22.7. The y-axis represents the number of errors that falls within each interval on the x-axis. Here we can see that two errors are greater than 8. These represent the same two errors that we identified as outliers in Figure 22.6.

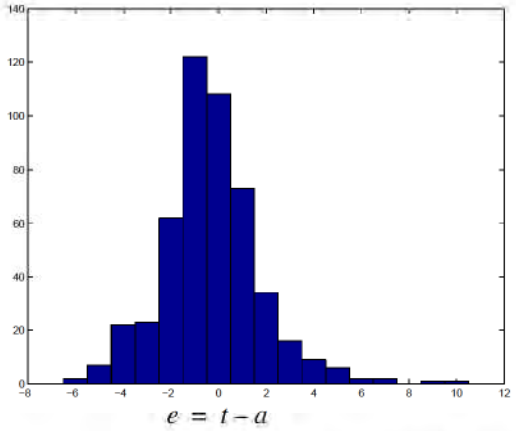


Figure 22.7 Histogram of Network Errors

Pattern Recognition

Confusion Matrix

For pattern recognition problems, the regression analysis is not as useful as it is for fitting problems, since the target values are discrete. However, there is an analogous tool - the *confusion (or misclassification) matrix*. The confusion matrix is a table whose columns represent the target class and whose rows represent the output class. For example, Figure 22.8 shows a sample confusion matrix in which there were 214 data points. There were 41 input vectors that belonged to Class 1 and were correctly classified as Class 1. There were 162 input vectors that belonged to Class 2 and were correctly classified as Class 2. The correctly classified inputs show in the diagonal cells of the confusion matrix. The off-diagonal cells show misclassified inputs. The lower left cell shows that four inputs from Class 1 were misclassified by the network as Class 2. If Class 1 is considered a positive outcome, then the lower left cell represents *false negatives*, which are also

False Negative

False Positive

called Type II errors. The upper right cell shows that one input from Class 2 was misclassified by the network as Class 1. This would be considered a *false positive* or a Type I error.

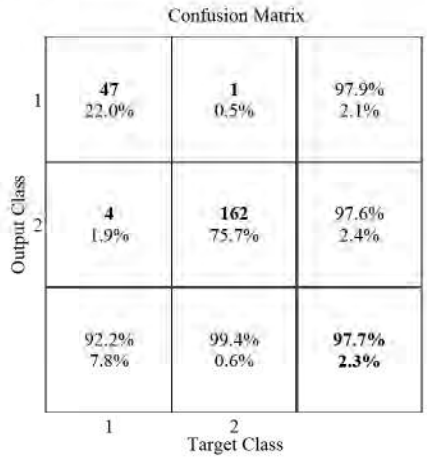


Figure 22.8 Sample Confusion Matrix

ROC Curve

Another useful tool for analyzing a pattern recognition network is called the *Receiver Operating Characteristic (ROC) curve*. To create this curve, we take the output of the trained network and compare it against a threshold which ranges from -1 to +1 (assuming a tansig transfer function in the last layer). Inputs that produce values above the threshold are considered to belong to Class 1, and those with values below the threshold are considered to belong to Class 2. For each threshold value, we count the fraction of true positives and false positives in the data set. This pair of numbers produces one point on the ROC curve. As the threshold is varied, we trace the complete curve, as shown in Figure 22.9.

The ideal point for the ROC curve to pass through would be (0,1), which would correspond to no false positives and all true positives. A poor ROC curve would represent a random guess, which is represented by the diagonal line in Figure 22.9, which passes through the point (0.5,0.5).

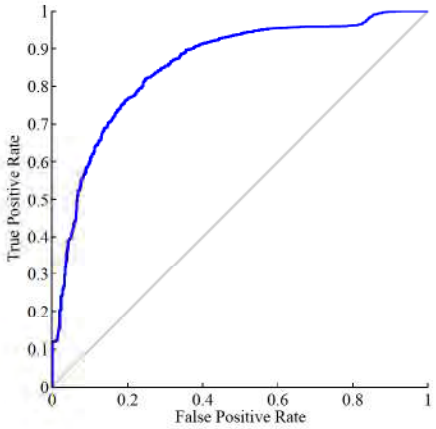


Figure 22.9 Receiver Operating Characteristic Curve

Clustering

Quantization Error

The SOM is the most commonly used network for clustering. There are several measures of SOM performance. One is *quantization error*. This is the average distance between each input vector and the closest prototype vector. It measures the map resolution. This can be made artificially small, if we use a large number of neurons. If there are as many neurons as input vectors in the data set, then the quantization error could be zero. This would represent overfitting. If the number of neurons is not significantly smaller than the number of input vectors, then the quantization error is not meaningful.

Topographic Error

Another measure of SOM performance is *topographic error*. This is the proportion of all input vectors for which the closest prototype vector and the next closest prototype vector are not neighbors in the feature map topology. Topographic error measures the preservation of the topology. In a well-trained SOM, prototypes that are neighbors in the topology should also be neighbors in the input space. In this case, the topographic error should be zero.

Distortion Measure

The performance of the SOM can also be assessed by the *distortion measure*:

$$E_d = \sum_{q=1}^{Q} \sum_{i=1}^{S} h_{ic_q} \|_i\mathbf{w} - \mathbf{p}_q\|^2, \tag{22.14}$$

where h_{ij} is the neighborhood function, and c_q is the index of the prototype that is closest to the input vector $\mathbf{p}_q$:

$$c_q = \arg\min_j \{ \| {}_j\mathbf{w} - \mathbf{p}_q \| \} . \tag{22.15}$$

For the simplest neighborhood function, h_{ij} is equal to 1 if prototype i is within some pre-specified neighborhood radius of prototype j, and equal to zero otherwise. It is also possible to have neighborhood functions that decrease continuously, such as the Gaussian function:

$$h_{ij} = \exp\left(\frac{-\| {}_i\mathbf{w} - {}_j\mathbf{w} \|^2}{2d^2} \right), \tag{22.16}$$

where d is the neighborhood radius.

Prediction

As we discussed earlier, one application of neural networks is the prediction of the future values of some time series. For prediction problems, we use dynamic networks, such as the focused time-delay neural network shown in Figure 22.3. There are two important concepts that are used when analyzing a trained prediction network:

1. the prediction errors should not be correlated in time, and
2. the prediction errors should not be correlated with the input sequence.

If the prediction errors were correlated in time, then we would be able to predict the prediction errors and, therefore, improve our original prediction. Also, if the prediction errors were correlated with the input sequence, then we would also be able to use this correlation to predict the errors.

Autocorrelation Function

In order to test the correlation of the prediction errors in time, we can use the sample *autocorrelation function*:

$$R_e(\tau) = \frac{1}{Q-\tau} \sum_{t=1}^{Q-\tau} e(t)e(t+\tau) . \tag{22.17}$$

White Noise

If the prediction errors are uncorrelated (*white noise*), then we would expect $R_e(\tau)$ to be close to zero, except when $\tau = 0$. To determine if $R_e(\tau)$ is close to zero, we can set an approximate 95% confidence interval [BoJe96] using the range

$$-\frac{2R_e(0)}{\sqrt{Q}} < R_e(\tau) < \frac{2R_e(0)}{\sqrt{Q}} . \tag{22.18}$$

We can say that $e(t)$ is white, if $R_e(\tau)$ satisfies Eq. (22.18) for $\tau \neq 0$. This concept is illustrated in Figure 22.10 and Figure 22.11. Figure 22.10 shows a sample autocorrelation function for the prediction errors of a network that has not been adequately trained. We can see that the autocorrelation

function does not fall totally within the bounds defined by Eq. (22.18), which are indicated by the dashed lines in the figure. Figure 22.11 shows the corresponding autocorrelation function when a network has been successfully trained. $R_e(\tau)$ falls within the bounds, except at $\tau = 0$.

Correlation in the prediction errors can indicate that the length of the tapped delay lines in the network should be increased.

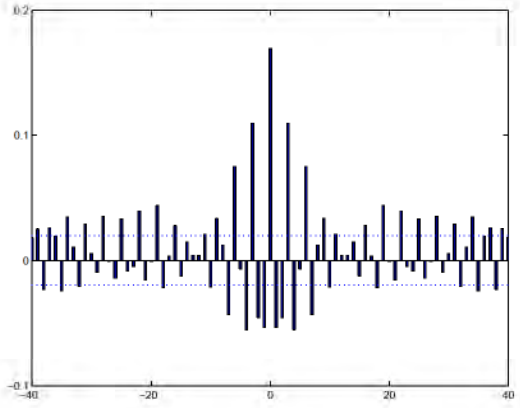


Figure 22.10 $R_e(\tau)$ for Inadequately Trained Network

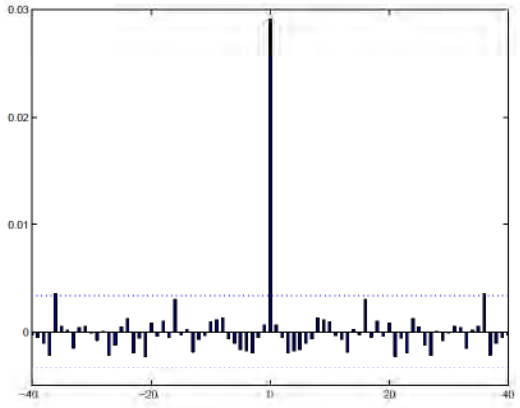


Figure 22.11 $R_e(\tau)$ for Successfully Trained Network

Cross-correlation Function

To test the correlation between the prediction errors and the input sequence, we can use the sample *cross-correlation function*:

$$R_{pe}(\tau) = \frac{1}{Q-\tau}\sum_{t=1}^{Q-\tau} p(t)e(t+\tau). \tag{22.19}$$

If there is no correlation between the prediction errors and the input sequence, then we would expect $R_{pe}(\tau)$ to be close to zero for all τ. To determine if $R_{pe}(\tau)$ is close to zero, we can set an approximate 95% confidence interval [BoJe96] using the range

$$-\frac{2\sqrt{R_e(0)}\sqrt{R_p(0)}}{\sqrt{Q}} < R_{pe}(\tau) < \frac{2\sqrt{R_e(0)}\sqrt{R_p(0)}}{\sqrt{Q}}. \tag{22.20}$$

This concept is illustrated in Figure 22.12 and Figure 22.13. Figure 22.12 shows a sample cross-correlation function for the prediction errors of a network that has not been adequately trained. We can see that the cross-correlation function does not fall totally within the bounds defined by Eq. (22.20), which are indicated by the dashed lines. Figure 22.13 shows the corresponding cross-correlation function when a network has been successfully trained. $R_{pe}(\tau)$ falls within the bounds for all τ.

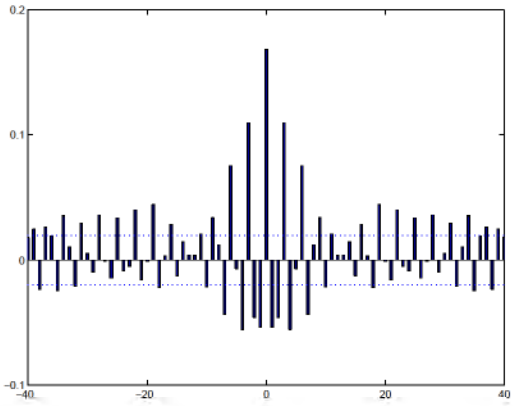


Figure 22.12 $R_{pe}(\tau)$ for Inadequately Trained Network

When using a NARX network, correlation between the prediction error and the input can suggest that the lengths of the tapped delay lines in the input and feedback paths should be increased.

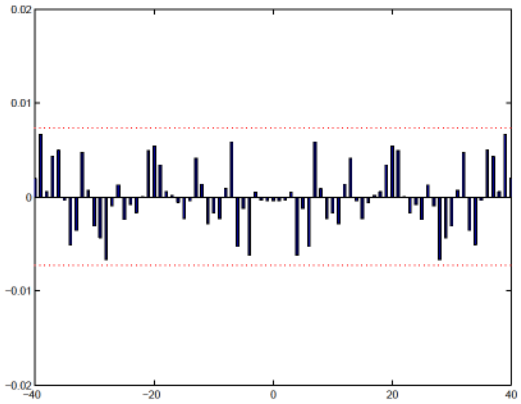


Figure 22.13 $R_{pe}(\tau)$ for Successfully Trained Network

Overfitting and Extrapolation

Recall from Chapter 13 that the total data set is divided into three parts: training, validation and testing. The training set is used to calculate gradients and to determine weight updates. The validation set is used to stop training before overfitting occurs. (If Bayesian regularization is used, then the validation set may be merged with the training set.) The test set is used to predict future performance of the network. The test set performance is the measure of network quality. If, after a network has been trained, the test set performance is not adequate, then there are usually four possible causes:

- the network has reached a local minimum,
- the network does not have enough neurons to fit the data,
- the network is overfitting, or
- the network is extrapolating.

The local minimum problem can almost always be overcome by retraining the network with five to ten random sets of initial weights. The network with minimum training error will generally represent a global minimum. The other three problems can generally be distinguished by analyzing the training, validation and test set errors. For example, if the validation error is much larger than the training error, then overfitting has probably occurred. Even though early stopping is used, it is possible to have some overfitting, if the training occurs too quickly. In this case, we can use a slower training algorithm to retrain the network.

If the validation, training and test errors are all similar in size, but the errors are too large, then it is likely that the network is not powerful enough to fit the data. In this case, we should increase the number of neurons in the hidden layer and retrain the network. If Bayesian regularization is used, this situation is indicated by the effective number of parameters becoming equal to the total number of parameters. When the network is large enough, the effective number of parameters should remain below the total number of parameters.

If the validation and training errors are similar in size, but the test errors are significantly larger, then the network may be extrapolating. This indicates that the test data fall outside the range of the training and validation data. In this case, we need to get more data. You can merge the test data into the training/validation data and then collect new test data. You should continue to add data until the results on all three data sets are similar.

If training, validation and test errors are similar, and the errors are small enough, then we can put the multilayer network to use. However, we still need to be careful about the possibility of extrapolation. If the multilayer network inputs are outside the range of the data with which it was trained, then extrapolation will occur. It is difficult to guarantee that training data will encompass all future uses of a neural network.

One method for detecting extrapolation is to train a companion competitive network to cluster the input vectors in the multilayer network training set. Then, when an input is applied to the multilayer network, the same input is applied to the companion competitive network. When the distance of the input vector to the nearest prototype vector of the competitive network is larger than the distance from the prototype to the most distant member of its cluster of inputs in the training set, we can suspect extrapolation. This technique is referred to as *novelty detection*.

Novelty Detection

Sensitivity Analysis

After a multilayer network has been trained, it is often useful to assess the importance of each element of the input vector. If we can determine that a given element of the input vector is unimportant, then we can eliminate it. This can simplify the network, reduce the amount of computation and help prevent overfitting. There is no one method that can absolutely determine the importance of each input, but a sensitivity analysis can be helpful in this regard. A sensitivity analysis computes the derivatives of the network response with respect to each element of the input vector. If the derivative with respect to a certain input element is small, then that element can be eliminated from the input vector.

Because the multilayer network is nonlinear, the derivative of the network output with respect to an input element will not be constant. For each input vector in the training set, the derivatives will be different. For this reason, we can't use a single derivative to determine sensitivity. One option would

be to take the average of the absolute derivatives, or else the rms derivatives, over the entire training set. Another option would be to compute the derivative of the sum square error with respect to each element of the input vector. Each of these will compute a single derivative for each element of the input vector. The last computation can be performed with a simple variation of the backpropagation algorithm (see Eq. (11.44) to Eq. (11.47)). Recall from Eq. (11.32) that

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}, \tag{22.21}$$

where $\hat{F}$ is a single error squared. We want to convert this to a derivative with respect to an element of the input vector, using the chain rule:

$$\frac{\partial \hat{F}}{\partial p_j} = \sum_{i=1}^{S^1} \frac{\partial \hat{F}}{\partial n_i^1} \times \frac{\partial n_i^1}{\partial p_j} = \sum_{i=1}^{S^1} s_i^1 \times \frac{\partial n_i^1}{\partial p_j}. \tag{22.22}$$

We know that

$$n_i^1 = \sum_{j=1}^{R} w_{i,j}^1 p_j + b_i^1, \tag{22.23}$$

therefore, Eq. (22.22) becomes

$$\frac{\partial \hat{F}}{\partial p_j} = \sum_{i=1}^{S^1} \frac{\partial \hat{F}}{\partial n_i^1} \times \frac{\partial n_i^1}{\partial p_j} = \sum_{i=1}^{S^1} s_i^1 \times w_{i,j}^1. \tag{22.24}$$

In matrix form, we can write this as

$$\frac{\partial \hat{F}}{\partial \mathbf{p}} = (\mathbf{W}^1)^T \mathbf{s}^1. \tag{22.25}$$

This will be the derivative for a single squared error. To get the derivative of the sum square error, we sum the individual derivatives for each single squared error. The resulting vector will contain the derivatives of the sum square error for each element of the input vector. If we find that some of these derivatives are much smaller than the maximum derivative, then we can consider removing those inputs. After removing the potentially irrelevant inputs, we retrain the network and compare the performance with the original network. If the performance is similar, then we accept the simplified network.

Epilogue

While previous chapters have focused on the fundamentals of particular network architectures and training rules, this chapter has discussed some practical aspects of neural network training. Neural network training is an iterative process involving data collection and preprocessing, network architecture selection, network training and post-training analysis.

The next five chapters will demonstrate some of these practical aspects, as we present some real-world case studies. The case studies will cover a variety of applications, including function fitting, density estimation, pattern recognition, clustering and prediction.

Further Reading

[Bish95] C.M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press,1995.

This well-written and well-organized textbook presents neural networks from a statistical perspective.

[BoJe94] G.E.P. Box, G.M. Jenkins, and G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, 4th Edition, John Wiley & Sons, 2008.

This is a classic text on time series analysis. It focuses on practical aspects, rather than theoretical derivations.

[HaBo07] L. Hamm, B. W. Brorsen and M. T. Hagan, "Comparison of Stochastic Global Optimization Methods to Estimate Neural Network Weights," *Neural Processing Letters*, Vol. 26, No. 3, December 2007.

This paper demonstrates that using multiple restarts of a local optimization procedure, like steepest descent or conjugate gradient, produces results that are comparable to global optimization methods, but with less computation.

[HeOh97] B. Hedén, H. Öhlin, R. Rittner, L. Edenbrandt, "Acute Myocardial Infarction Detected in the 12-Lead ECG by Artificial Neural Networks," *Circulation*, vol. 96, pp. 1798–1802, 1997.

Describes the use of neural networks in detecting myocardial infarctions, using the electrocardiogram.

[Joll02] I.T. Jolliffe, *Principal Component Analysis*, Springer Series in Statistics, 2nd ed., Springer, NY, 2002.

The most popular text on principal component analysis.

[LeCu98] Y. LeCun, L. Bottou, G. B. Orr, K.-R. Mueller, "Efficient BackProp," *Lecture Notes in Comp. Sci.*, vol. 1524, 1998.

This paper presents practical tips that improve the training of multilayer networks.

[Moll93] M. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, pp. 525-533, 1993.

The scaled conjugate gradient algorithm presented in this paper converges quickly, and with a minimum amount of memory requirements.

[NgWi90] D. Nguyen and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the IJCNN*, vol. 3, pp. 21–26, July 1990.

This paper describes a procedure for setting the initial weights and biases for the backpropagation algorithm. It uses the shape of the sigmoid transfer function and the range of the input variables to determine how large the weights should be, and then uses the biases to center the sigmoids in the operating region. The convergence of backpropagation is improved significantly by this procedure.

[PeCo93] M. P. Perrone and L. N. Cooper, "When networks disagree: Ensemble methods for hybrid neural networks," in *Neural Networks for Speech and Image Processing*, R. J. Mammone, Ed., Chapman-Hall, pp. 126-142, 1993.

This paper describes how you can combine the outputs of a committee of networks to produce results that are more accurate than any of the individual networks.

[PuFe97] G.V. Puskorius and L.A. Feldkamp, "Extensions and enhancements of decoupled extended Kalman filter training," *Proceedings of the 1997 International Conference on Neural Networks*, vol. 3, pp. 1879-1883, 1997.

The extended Kalman filter algorithm described in this paper is one of the faster sequential algorithms for neural network training.

[RaMa05] L.M. Raff, M. Malshe, M. Hagan, D.I. Doughan, M.G. Rockley, and R. Komanduri, "*Ab initio* potential-energy surfaces for complex, multi-channel systems using modified novelty sampling and feedforward neural networks," *The Journal of Chemical Physics*, vol. 122, 2005.

This paper describes how neural networks can be used to model molecular interactions.

[ScSm99] B. Schölkopf, A. Smola, K.-R. Muller, "Kernel Principal Component Analysis," in B. Schölkopf, C. J. C. Burges, A. J. Smola (Eds.), *Advances in Kernel Methods-Support Vector Learning*, MIT Press Cambridge, MA, USA, pp. 327-352, 1999.

This paper introduces a nonlinear version of principal component analysis using a kernel method.

23 Case Study 1: Function Approximation

Objectives	23-1
Theory and Examples	23-2
Description of the Smart Sensor System	23-2
Data Collection and Preprocessing	23-3
Selecting the Architecture	23-4
Training the Network	23-5
Validation	23-7
Data Sets	23-10
Epilogue	23-11
Further Reading	23-12

Objectives

This chapter represents the first of a series of case studies with neural networks. Neural networks can be used for a wide variety of applications, and it would be impossible to provide case studies for each application. We will limit our presentations to five important application areas: function approximation (aka, nonlinear regression), density function estimation, pattern recognition (aka, pattern classification), clustering and prediction (aka, time series analysis, system identification, or dynamic modeling). For each case study, we will step through the neural network design/training process.

In this chapter, we present a function approximation problem. For function approximation problems, the training set consists of a set of dependent variables (response variables) and one or more independent variables (explanatory variables). The neural network learns to create a mapping between the explanatory variables and the response variables. In the case study we consider in this chapter, the system in question is a smart sensor. A smart sensor consists of one or more standard sensors that are coupled with a neural network to produce a calibrated measurement of a single parameter. In this chapter, we will consider a smart position sensor, which uses the voltages coming from two solar cells to produce an estimate of the location of an object in one dimension.

Theory and Examples

This chapter presents a case study in using neural networks for function approximation. Function approximation consists of defining a mapping between a set of input variables and a corresponding set of output variables. For example, we might want to estimate the price of a home, based on characteristics of the neighborhood, such as tax rate, pupil/teacher ratio in local schools and crime rate. Another example would be estimating octane number of a gasoline product at an oil refinery, based on measurements of reactor temperatures and pressures [FoGi07]. In the case study presented in this chapter, we will consider a smart position sensor system.

Description of the Smart Sensor System

Figure 23.1 illustrates the sensor arrangements for this case study. An object is suspended between a light source and two solar cells. The object casts a shadow on the solar cells, which causes the voltage out of the solar cells to decrease.

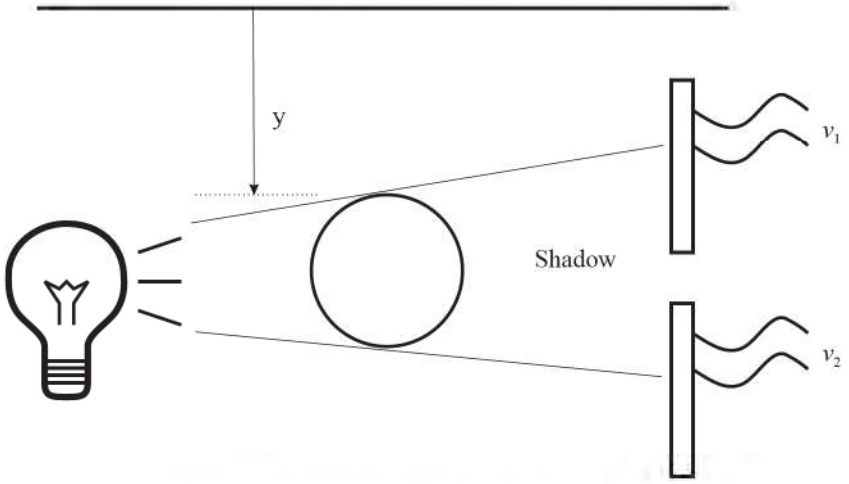


Figure 23.1 Position Sensor Arrangement

As the object position y increases, first the voltage v_1 decreases, then the voltage v_2 decreases, then v_1 increases, and finally v_2 increases. This is demonstrated in Figure 23.2. Our objective is to determine the object position from measurements of the two voltages. Clearly this is a very nonlinear relationship, so a multilayer network will be needed to learn the mapping. This is a classic type of function approximation problem, in which we are trying to learn the inverse of a function. The forward function is the

mapping from y to v_1 and v_2. We want to learn the mapping from v_1 and v_2 to y.

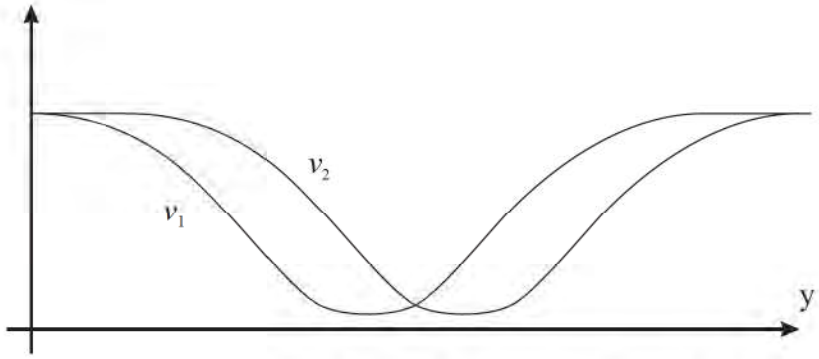


Figure 23.2 Example Solar Cell Outputs vs. Object Position

Data Collection and Preprocessing

In order to collect data for this process, we took measurements of the two solar cell voltages at a number of calibrated positions of the object. The object we used for these experiments was a table tennis ball. The data is displayed in Figure 23.3. There are a total of 67 sets of measurements. Each circle represents a voltage measurement at a calibrated position. The units of position are inches, and the units of voltage are volts. The flat regions at 0 volts for each curve occur where the shadow of the ball completely covers a sensor. If the shadow were large enough to cover both cells at the same time, we would not be able to recover ball position from cell voltages.

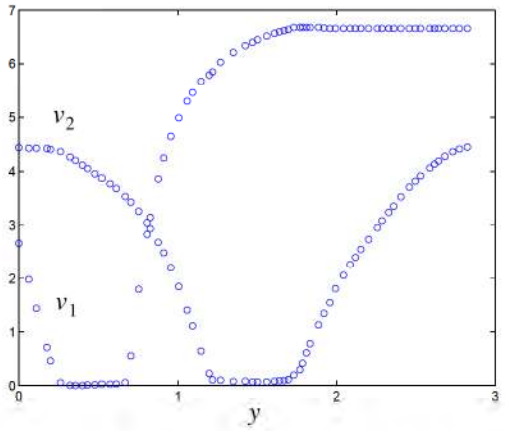


Figure 23.3 Data Collected from Solar Cells

The next step is to divide the data into training, validation and test sets. In this case, because we will be using the Bayesian regularization training technique, we do not need to have a validation set. We did set aside 15% of the data for testing purposes. To perform the division, we arranged the data in order, according to object position, and then selected every sixth or seventh point for testing. This resulted in 10 testing points. The testing points are not used in any way for training the network, but after the network has been completely trained, we will use the testing data as an indicator of future network performance.

The input vector for network training will consist of the solar cell voltages

$$\mathbf{p} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \tag{23.1}$$

and the target will be ball position

$$t = y. \tag{23.2}$$

The data were scaled using Eq. (22.1), so that both the inputs and the targets were in the range [-1,1]. The resulting scaled data is shown in Figure 23.4

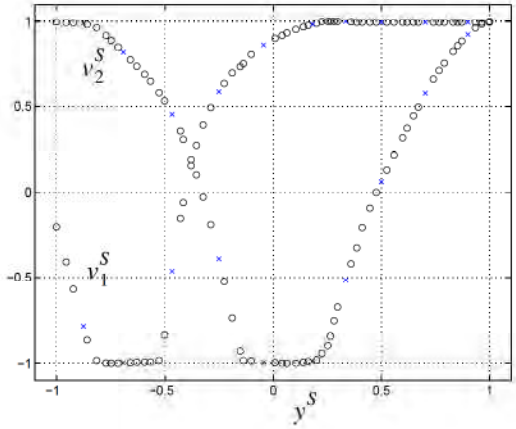


Figure 23.4 Scaled Data

Selecting the Architecture

Because the mapping between the solar cell voltages and the ball position is highly nonlinear, we will use a multilayer network architecture to learn the mapping. We know that there will be two elements in the input vector,

which is defined in Eq. (23.1). The single target for the network is the ball position, given in Eq. (23.2).

Figure 23.5 shows the network architecture. We are using the tan-sigmoid transfer function in the hidden layer, and a linear output layer. This is the standard network for function approximation. As we discussed in Chapter 11, this network has been shown to be a universal approximator. There are cases in which two hidden layers are used, but we normally try first with one hidden layer. The number of neurons in the hidden layer, S^1, will depend on the function to be approximated. This is something that cannot generally be known before training. We will have more to say about this in the next section.

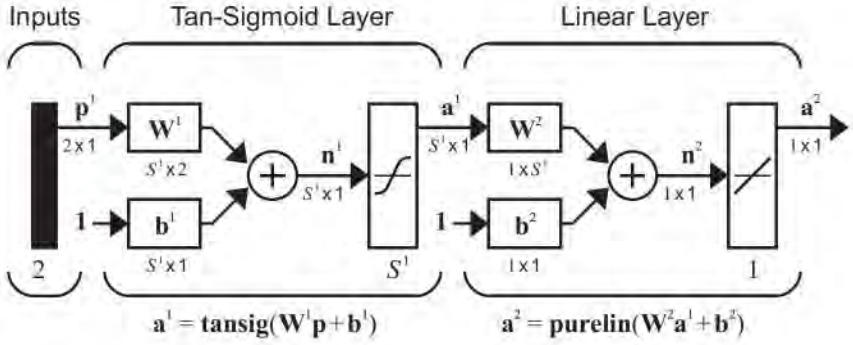


Figure 23.5 Network Architecture

Training the Network

Before beginning the training, we initialized the network weights using the method of Widrow and Nguyen described in Chapter 22. Then we used Bayesian regularization to train the network. Bayesian regularization, which we discussed in Chapter 13, is a very effective algorithm for training multilayer networks to perform function approximation. This algorithm is designed to train networks so that they generalize well, without the need for a validation set. Because the validation set can then be added to the training set, the performance is often better than that obtained with early stopping. (In the next chapter, we will give an example of using early stopping, with a validation set.)

Figure 23.6 illustrates the sum square error versus iteration number, while using the Bayesian regularization training algorithm. We used a network with 10 neurons in the hidden layer ($S^1 = 10$) for this case. The network was trained for 100 iterations, at which time the performance was changing very little.

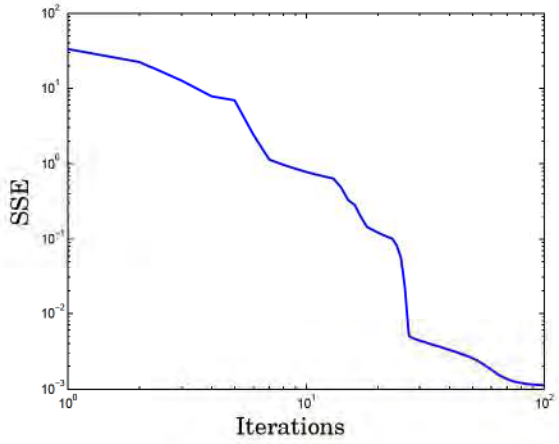


Figure 23.6 Sum Squared Error vs. Iteration Number ($S^1 = 10$)

Training has converged after 100 iterations, but we want to ensure that we have not fallen into a local minimum. For this reason, we want to retrain the network several times, using different initial weights and biases. (We use the Nguyen-Widrow initialization method described in Chapter 22.) Table 23.1. shows the final validation SSE for each of five different training runs. We can see that all of the errors are similar, although the errors are slightly smaller for runs 2, 4 and 5. Any of the weights from these five cases would produce a satisfactory network. We will discuss this in more detail in the next section.

1.121e-003	8.313e-004	1.068e-003	8.672e-004	8.271e-004

Table 23.1. Final Training SSE for Five Different Initial Conditions

Recall from Chapter 13 that the Bayesian regularization algorithm computes a parameter γ, which indicates the effective number of parameters that are being used by the network. In Figure 23.7, we can see the variation of γ during training. It eventually converges to 17.4. There are a total of 41 parameters in this 2-10-1 network, so we are only using about 40% of the weights and biases. For each of the five training runs discussed above, γ converged to values between 17 and 20. This indicates that we might be able to use a smaller network, if we are concerned about the amount of computation required to compute a network response.

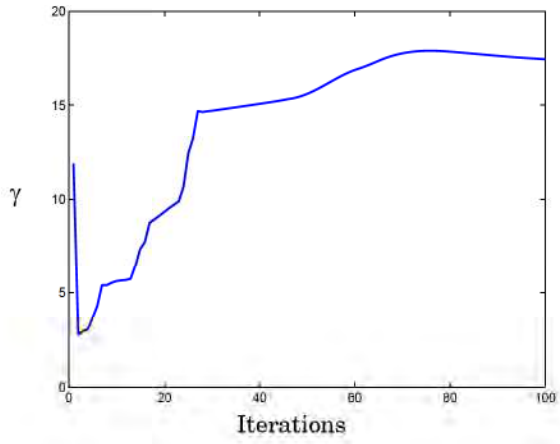


Figure 23.7 Effective Number of Parameters ($S^1 = 10$)

To see whether or not a smaller network would be satisfactory, we trained several networks with different numbers of hidden neurons. Since the effective number of parameters is near 20, we would expect that a network with five hidden neurons (21 weights and biases) might provide an adequate fit. Our experiments produced the results shown in Table 23.2. We can see that the performances of all five networks are roughly equivalent, except for the case where $S^1 = 3$, where the total number of parameters in the network is only 13.

$S^1 = 3$	$S^1 = 5$	$S^1 = 8$	$(S^1 = 10)$	$S^1 = 20$
4.406e-003	9.227e-004	8.088e-004	8.672e-004	8.096e-004

Table 23.2. Final Training SSE for Five Different Hidden Layer Sizes

The Bayesian regularization method allows us to train a network of almost arbitrary size, and yet insure that only the required number of parameters is effectively used. If we were concerned about the amount of time required to compute the network output (e.g., for real-time applications), then we would want to use the network with $S^1 = 5$. Otherwise, the original network with $S^1 = 10$ is satisfactory. We don't need to spend a lot of time finding the optimal number of neurons. The training algorithm will insure that we do not overfit.

Validation

An important tool for network validation is a scatter plot of network outputs versus targets, as shown in Figure 23.8 (in normalized units). We ex-

pect that for a well trained network the points in the scatter plot will fall close to the 45° output=target line. In this case, the fit is excellent. The figure on the left shows the training data, while the figure on the right shows the testing data. Because the testing data fit is as good as the training data fit, we can be confident that the network did not overfit.

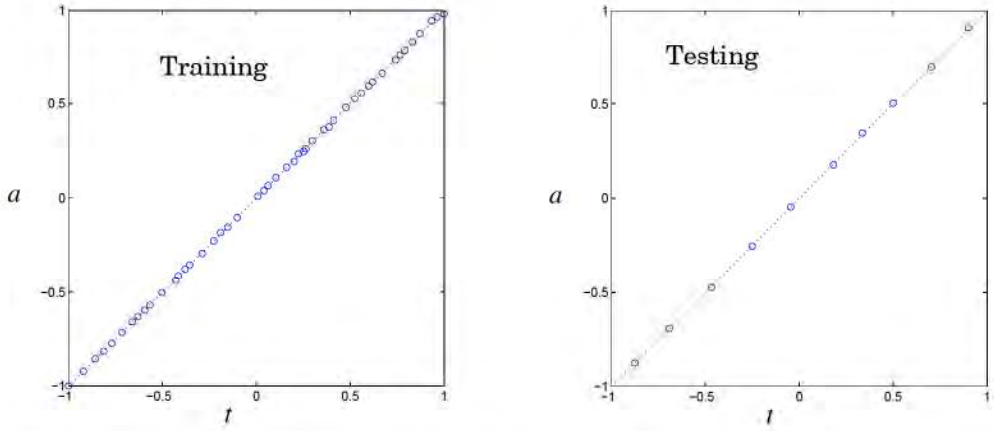


Figure 23.8 Scatter Plots of Network Outputs vs. Targets - Training and Testing Sets

Another useful plot is a histogram of network error, as shown in Figure 23.9. This gives us an idea of the accuracy of the network. For this histogram, we have converted the network output back into units of inches. This is done by applying the reverse of the target preprocessing function to the network output. The reverse of Eq. (22.1), for the targets, is given by

$$\mathbf{a} = (\mathbf{a}^n + 1).*\frac{(\mathbf{t}^{max} - \mathbf{t}^{min})}{2} + \mathbf{t}^{min} \tag{23.3}$$

where $\mathbf{a}^n$ is the original network output, which was trained to match the normalized target, and .* represents an element-by-element multiplication of two vectors. After the postprocessing operation of Eq. (23.3), the resulting un-normalized output is subtracted from the raw targets, to produce an error in inches. Figure 23.9 shows the distribution of these errors for both training and testing sets. We can see that almost all errors are within one hundredth of an inch. This is within the accuracy of the original measurements, so we cannot expect to do better.

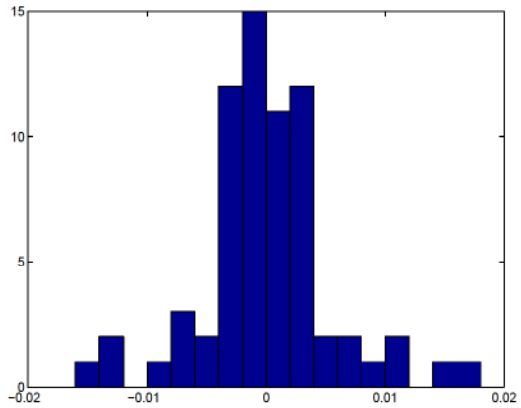


Figure 23.9 Histogram of Position Errors (in Inches)

Because this network has only two inputs, we can plot the trained network response, which is shown in Figure 23.10. (This figure shows the response from original unscaled inputs, in volts, to original unscaled output, in inches.) The blue circles indicate the path that is taken by the voltages as the ball is moved. Notice that the Bayesian regularization training has produced a smooth network response, even though the response is highly nonlinear.

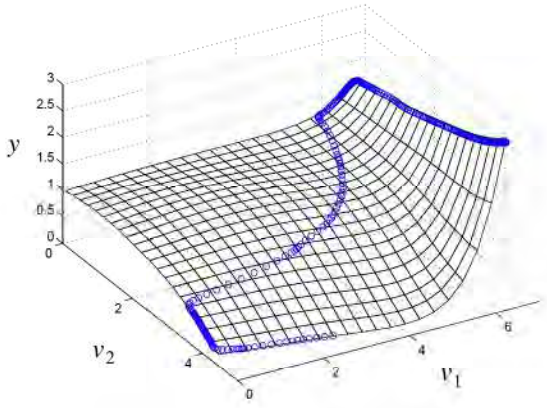


Figure 23.10 Network Response (Original Units)

Another thing to notice about Figure 23.10 is that training data only falls along the blue circles. The form of the network response in other regions is of no importance for the operation of the smart sensor system, since the

network will never be used there. If the network were retrained, the shape of the response away from the blue circles might be very different, even though the response near the blue circles will always be the same. This concept is very important for many neural network applications. Often, during normal network operation, only a small portion of the input space will be accessed. The network only has to fit the underlying function in these regions where the network will be used. This means that the size of the data set can be modest, even when the input dimension is large. Of course, in these cases, it is critical that the training data span the full range of potential network operation.

Data Sets

There are two data files associated with this case study:

- ball_p.txt — contains the input vectors in the original data set
- ball_t.txt — contains the target vectors in the original data set

They can be found with the demonstration software, which is described in Appendix C.

Epilogue

This chapter has demonstrated the use of multilayer neural networks for function approximation. This case study is representative of a large class of neural network applications that could be termed "soft sensors" or "smart sensors." The idea is to use a neural network to fuse several raw sensor outputs into a calibrated measurement of some key variable of interest.

A multilayer network with sigmoid transfer functions in the hidden layers and linear transfer functions in the output layer is well suited to this type of application, and Bayesian regularization is an excellent training algorithm to use in this situation.

In the next chapter, we will look at another neural network application — probability estimation. We will also use multilayer neural networks for that application, but we will change the transfer function in the output layer.

Further Reading

[FoGi07] L. Fortuna, P. Giannone, S. Graziani, M. G. Xibilia, "Virtual Instruments Based on Stacked Neural Networks to Improve Product Quality Monitoring in a Refinery," *IEEE Transactions on Instrumentation and Measurement*, vol. 56, no. 1, pp. 95–101, 2007.

This paper describes the use of neural networks as soft sensors in a refinery. Measurements of reactor temperatures and pressures are used to predict octane number in a gasoline product.

24 Case Study 2: Probability Estimation

Objectives	24-1
Theory and Examples	24-2
Description of the CVD Process	24-2
Data Collection and Preprocessing	24-3
Selecting the Architecture	24-5
Training the Network	24-7
Validation	24-9
Data Sets	24-12
Epilogue	24-13
Further Reading	24-14

Objectives

This chapter represents the second of a series of case studies with neural networks. The previous chapter demonstrated the use of neural networks for function approximation. In this chapter we use a neural network to estimate a probability function.

Probability estimation is a special case of function approximation. In function approximation we want the neural network to map between a set of input variables and a set of response variables. However, in the case of probability estimation the response variables correspond to a set of probabilities. Since probabilities have certain special properties — they must always be positive, and they must sum to 1 — we want the neural network to enforce these conditions.

In the case study we consider in this chapter, the system in question is chemical vapor deposition of diamond. A carbon dimer (a bound pair of carbon atoms) is projected toward a diamond surface. We want to determine the probabilities for various reactions based on characteristics of the projected dimer. The input variables consist of such properties as translational energy and incidence angle, and the response variables consist of the probabilities of the potential reactions, such as chemisorption and scattering.

Theory and Examples

This chapter presents a case study in using neural networks for probability estimation. Probability estimation consists of determining the probabilities of certain events, based on a set of input variables. For example, we might want to know the probabilities associated with a patient having a certain disease, based on a set of laboratory tests. Another example would be determining the probability of a financial instrument going up in price, based on a set of market conditions.

CVD

For this probability estimation case study, we will train a neural network to estimate reaction rates in a chemical process. Chemical vapor deposition (CVD) of diamond is a process for making synthetic diamond. The idea is to cause carbon atoms in a gas to settle on a substrate in crystalline form. In order to study this process, scientists are often interested in reaction rates, which will determine how quickly the diamond can be created. In this case study, we will train a neural network to compute reaction rates as a carbon dimer (a bound pair of carbon atoms) interacts with the crystalline diamond substrate.

We will begin by describing the CVD process and how simulated data can be collected for this process. Then, we will show how a neural network can be trained to learn the reaction probabilities. The details of the procedure are described in [AgSa05].

Description of the CVD Process

During the CVD process, a carbon dimer is projected toward a diamond substrate. For the purpose of this study, we will assume that the dimer can react with the substrate in one of three ways: chemisorption (the atoms in the dimer become bound to the substrate), scattering (the atoms bounce off the substrate), or desorption (the atoms become bound to the substrate for a period of time, but are then released). There is another possible reaction that occurs with very small probability, but we will ignore it for this study. (See [AgSa05] for a full discussion.) We will train a neural network to estimate the probabilities of each of the reactions, based on various characteristics of the carbon dimer, which will be described below.

The notation we will use to define this interaction is illustrated in Figure 24.1. The black circle represents the carbon dimer, and the corresponding directed line represents the direction of the initial velocity vector. The blue star represents the location of the central carbon atom in the diamond substrate. The angle θ denotes the angle of incidence, i.e., the angle between the direction of the initial velocity vector of the carbon dimer and the perpendicular on the surface (the z direction). The impact parameter b is defined as the distance between the location of the central atom and the point of intersection of the initial velocity vector and the diamond surface (indi-

cated by the origin of the axes in Figure 24.1). The angle ϕ represents the angle between the x axis and the line from the origin to the central atom.

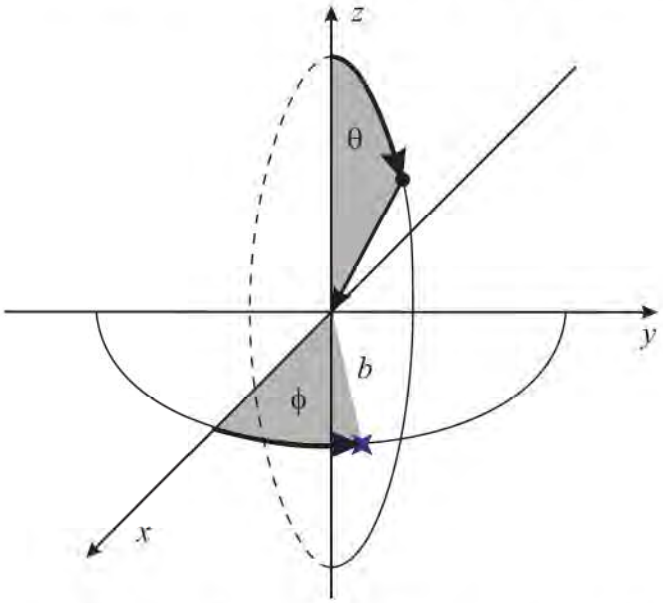


Figure 24.1 Notation for Carbon Dimer/Diamond Substrate Interaction

Data Collection and Preprocessing

Data for training the neural network are obtained by molecular dynamics (MD) simulations. In MD, the motion of atoms and molecules in a material under a given force are simulated, using known laws of physics to calculate the forces on individual atoms [RaMa05]. For this case study, we use a total of 324 atoms to model the CVD system. Out of these, 282 atoms of diamond substrate are used to model the crystalline face with 40 atoms of hydrogen on the top layer of the diamond surface, and 2 atoms in the C_2 dimer. In Figure 24.1, the (x,y) plane represents the location of the diamond substrate. Each of the carbon atoms on the top layer of the substrate, except the central atom and boundary atoms, is capped by a hydrogen atom. Any reactions will occur near the central atom.

For this study, we want to determine the dependence of probabilities for chemisorption, scattering, and desorption on b, θ, ϕ, rotational velocity (v_{rot}), and translational velocity (v_{trans}) of the C_2 dimer. The initial C_2 vibrational energy is set equal to the zero-point energy and the temperature of the lattice is maintained constant at 600 K [RaMa05].

This is an interesting function approximation problem, in that we don't have access to the true underlying reaction probabilities, which are un-

known. We will obtain estimates of these probabilities by running Monte Carlo simulation experiments. We will need some notation to help us keep track of the various probabilities that we will work with. First, we will indicate the true underlying reaction probabilities by $P_X(\mathbf{p})$, where X refers to the reaction process, and $\mathbf{p}$ is the vector that characterizes the C_2 dimer:

$$\mathbf{p} = \begin{bmatrix} \theta \\ \phi \\ b \\ v_{\text{trans}} \\ v_{\text{rot}} \end{bmatrix}, \tag{24.1}$$

The reaction process can be chemisorption ($X = C$), scattering ($X = S$), or desorption ($X = D$). The probability estimates produced by the neural network will be indicated by $P_X^{NN}(\mathbf{p})$. The probability estimates obtained from the Monte Carlo simulations will be indicated by $P_X^{MC}(\mathbf{p})$.

The Monte Carlo estimates are obtained by

$$P_X^{MC}(\mathbf{p}) = \frac{N_X}{N_T}, \tag{24.2}$$

where N_X is the number of MD trajectories that resulted in reaction X and N_T is the total number of trajectories computed. The results of a given trajectory depend upon a multitude of input variables. These include the parameters included in $\mathbf{p}$, as well as the initial orientation of the C_2 dimer, the angle defining the C_2 rotational plane, the initial C_2 vibrational energy and its phase, the temperature of the system, and all of the variables that define the vibrational phases of the diamond surface. Because we are only interested in the effect of $\mathbf{p}$ on the reaction probabilities, the other variables are randomly set for each MD simulation, except that the initial C_2 vibrational energy is set equal to the zero-point energy and the temperature of the lattice is maintained constant at 600 K. Eq. (24.2) averages over the trajectories to estimate the underlying true probabilities $P_X(\mathbf{p})$. (As a note of clarification here, we use the term Monte Carlo to refer to the set of simulations that are obtained by setting a number of the variables to random values for each trajectory. We refer to the simulation of a single trajectory as an MD simulation, since the principles of molecular dynamics are used to perform the computations.)

This is a standard method used by chemists to estimate reaction probabilities. If they want to determine the effect of ϕ, for example, on the probabilities, they must run a series of Monte Carlo simulations at each value of ϕ that is of interest. This can be extremely time consuming. The required number of Monte Carlo trials can be quite large, if an accurate reaction probability is required. Our objective in this case study is to train a neural

network to learn the true reaction probabilities as a function of the parameters in $\mathbf{p}$.

To train a neural network, we need a set of target outputs. Since we do not know the true underlying probabilities $P_X(\mathbf{p})$, we will use the estimates obtained from the Monte Carlo simulations $P_X^{MC}(\mathbf{p})$. We can think of these Monte Carlo probabilities as being noisy versions of the true probabilities. The neural network will need to interpolate these noisy values to produce an accurate estimate of $P_X(\mathbf{p})$ without overfitting. This is a good application for our generalization procedures, which were discussed in Chapter 13.

The data set consists of 2000 different $\{\mathbf{p}, P_X^{MC}(\mathbf{p})\}$ input/target pairs. Out of these 2000 data points, 1400 (70%) were randomly selected for training, 300 (15%) for validation, and 300 for testing. For each trajectory, the $\mathbf{p}$ were generated randomly, using physically-appropriate distributions for each variable [RaMa05]. A total of $N_T = 50$ different trajectories were run to obtain each $P_X^{MC}(\mathbf{p})$. This means that 2000x50 trajectories were run to create the entire data set.

The original units of the inputs are radians for ϕ and θ, angstroms for b, angstroms per picosecond for v_{trans} and radians per picosecond for v_{rot}. Before presenting the input data to the network for training, they are scaled using Eq. (22.1), so that each element of the input vector ranges from -1 to 1. The targets have values that are always in the range 0 to 1, since they represent probabilities. In the next section, we will describe a network architecture, in which the softmax transfer function of Eq. (22.3) is used in the final layer. This transfer function produces outputs that range from 0 to 1, so the original unscaled targets will work fine.

Selecting the Architecture

We will use a multilayer network for this application. We know that there will be five elements in the input vector, which is defined in Eq. (24.1). The target for the network can be a vector with three elements:

$$\mathbf{t} = \begin{bmatrix} P_C^{MC}(\mathbf{p}) \\ P_S^{MC}(\mathbf{p}) \\ P_D^{MC}(\mathbf{p}) \end{bmatrix}, \tag{24.3}$$

or we can use three different networks, each with a different $P_X^{MC}(\mathbf{p})$ as a target. We have tried both possibilities, and the results are similar.

In this case, there is an advantage to using the single network, with three elements in the output vector. The three targets represent probabilities. Therefore, they are always in the range 0 to 1, and they always sum to 1.

This is an ideal situation for using the softmax transfer function of Eq. (22.3), which is repeated here:

$$a_i = f(n_i) = \exp(n_i) \div \sum_{j=1}^{S} \exp(n_j). \tag{24.4}$$

This transfer function is different from others we have used, in that each neuron output a_i is affected by all of the net inputs n_j. (In the other transfer functions, the net input n_i affected only the neuron output a_i.) This does not cause any substantial difficulties in network training. The backpropagation algorithm of Eq. (11.44) and Eq. (11.45) can still be used to compute the gradient. However, the derivative of the transfer function is no longer a diagonal matrix. The derivative of the softmax function has the following form:

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} a_1^m\left(\sum_{i=1}^{S^m} a_i^m - a_1^m\right) & -a_1^m a_2^m & \cdots & -a_1^m a_{S_m}^m \\ -a_2^m a_1^m & a_2^m\left(\sum_{i=1}^{S^m} a_i^m - a_2^m\right) & \cdots & -a_2^m a_{S_m}^m \\ \vdots & \vdots & & \vdots \\ -a_{S_m}^m a_1^m & -a_{S_m}^m a_2^m & \cdots & a_{S_m}^m\left(\sum_{i=1}^{S^m} a_i^m - a_{S_m}^m\right) \end{bmatrix} \tag{24.5}$$

The complete network architecture is shown in Figure 24.2. The input vector of Eq. (24.1) has 5 elements. The output vector has 3 elements, which is consistent with the target vector of Eq. (24.3). The transfer function in the hidden layer is the hyperbolic tangent sigmoid, and the softmax transfer function is used in the output layer. The number of neurons in the hidden layer, S^1, is yet to be determined. It depends on the complexity of the function that we are trying to approximate, but we do not know at this point how complex the function is. In general, the size of the hidden layer must be determined as part of the training process. We must choose S^1 so that the network provides an accurate fit to the training data, without overfitting. We will discuss this selection in the next section.

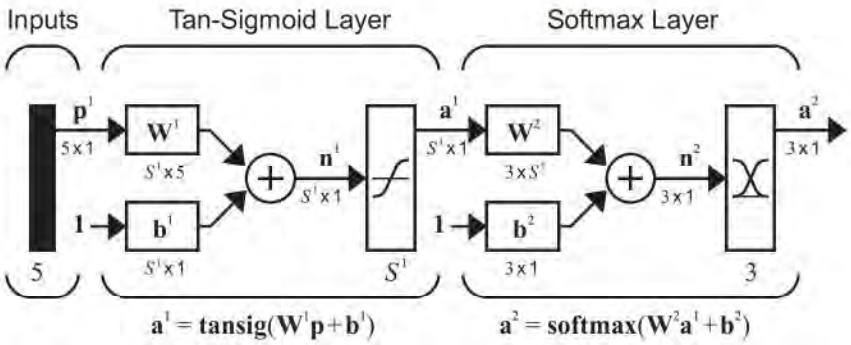


Figure 24.2 Network Architecture

Training the Network

We trained the network using the scaled conjugate gradient algorithm of [Mill93]. Many other conjugate gradient or Levenberg-Marquardt algorithms, such as those discussed in Chapter 12, would have also worked well. The targets for this problem have a significant amount of noise, so we are not expecting extreme accuracy in the final fit. We used early stopping, as described in Chapter 13, to prevent overfitting. We stopped the training if the error on the validation set failed to improve over 25 iterations. A typical training session is illustrated in Figure 24.3, which shows training and validation MSE. The minimum of the validation performance was reached at iteration 69. The algorithm continued for 25 more iterations, until iteration 94. Since the validation error was not reduced during those 25 iterations, the weights from iteration 69 were saved as the final trained values.

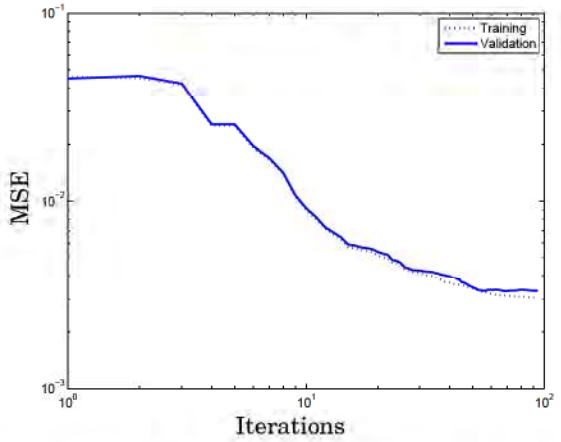


Figure 24.3 Training and Validation Mean Square Error ($S^1 = 10$)

The results shown in Figure 24.3 represent a network with 10 neurons in the hidden layer ($S^1 = 10$). We need to verify that this is a reasonable number. One indicator is a comparison of training and validation performance. Table 24.1. shows the training and validation root mean square error (RMSE) for the trained network. We can see that training and validation errors are roughly the same. The validation data was randomly selected and was selected independently of the training set. Because the errors were approximately the same on both sets, it appears that the network fit is consistent throughout the relevant input range, and no overfitting occurs.

	Training RMSE	Validation RMSE
$P_C(\mathbf{p})$	0.0496	0.0439
$P_S(\mathbf{p})$	0.0634	0.0659
$P_D(\mathbf{p})$	0.0586	0.0604

Table 24.1. Comparison of Training and Validation RMSE for $S^1 = 10$

It is also important to determine if the errors are as small as possible and if the fit is adequate. We will have more to say about that in the next section, but at this point we can try fitting networks with different numbers of hidden neurons. Table 24.2. shows the results of fitting a network with two hidden neurons. Again, the training and validation errors are consistent, which indicates lack of overfitting, but the errors are higher than those for $S^1 = 10$.

	Training RMSE	Validation RMSE
$P_C(\mathbf{p})$	0.0634	0.0627
$P_S(\mathbf{p})$	0.0669	0.0704
$P_D(\mathbf{p})$	0.0617	0.0618

Table 24.2. Comparison of Training and Validation RMSE for $S^1 = 2$

Table 24.3. shows the results for $S^1 = 20$. The validation error is slightly higher than the training error, which might indicate some overfitting. The main point is that neither training nor validation errors are significantly smaller for $S^1 = 20$ than for $S^1 = 10$. This indicates that ten hidden neurons are sufficient for this problem. We will investigate this further in the next section.

	Training RMSE	Validation RMSE
$P_C(\mathbf{p})$	0.0432	0.0444
$P_S(\mathbf{p})$	0.0603	0.0643
$P_D(\mathbf{p})$	0.0569	0.0595

Table 24.3. Comparison of Training and Validation RMSE for $S^1 = 20$

There is one further step that we want to make as part of the training process. We want to ensure that we have not fallen into a local minimum. For this reason, we want to retrain the network several times, using different initial weights and biases. (We use the Nguyen-Widrow initialization method described in Chapter 22.) Table 24.4. shows the final validation MSE for each of five different training runs. We can see that all of the errors are similar, so we have reached a global minimum at each run. If one error was significantly lower than the others, then we would use the weights that obtained the lowest error.

3.074e-003	2.953e-003	3.031e-003	3.105e-003	3.050e-003

Table 24.4. Final Validation MSE for Five Different Initial Conditions

We have determined that a neural network with ten neurons in the hidden layer produces a reasonable response without overfitting. The next step is to analyze the performance of the network. Depending on the results of that analysis, we might adjust the network architecture or training data and retrain the network.

Validation

An important tool for network validation is a scatter plot of network outputs versus targets, as shown in Figure 24.4. Here we can see that there is a strong linear relationship between the targets and the network outputs, but there appears to be quite a bit of variation. We might expect that for a well trained network the points in the scatter plot would fall exactly on the outputs=target line. Why do we have so much variation in this plot? The reason is that the targets of the network are not the true reaction probabilities, $P_X(\mathbf{p})$, but the Monte Carlo estimates $P_X^{MC}(\mathbf{p})$. There is noise in the targets.

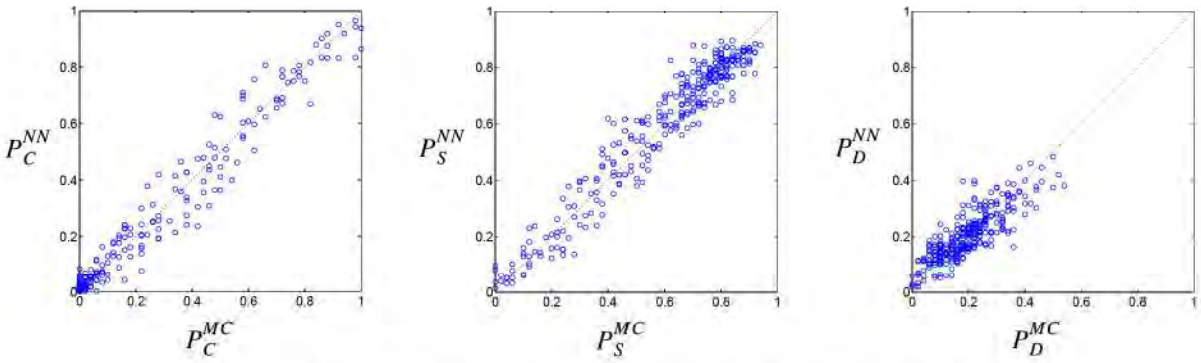


Figure 24.4 Scatter Plot of Network Outputs vs. Targets ($N_T = 50$)

The relationship between P_X^{MC} and P_X is such that ~95% of the time we expect to have

$$P_X - 2\Delta \le P_X^{MC} \le P_X + 2\Delta\,, \tag{24.6}$$

where

$$\Delta = \sqrt{\frac{P_X\{1 - P_X\}}{N_T}}\,. \tag{24.7}$$

This relationship is illustrated in Figure 24.5 for $N_T = 50$.

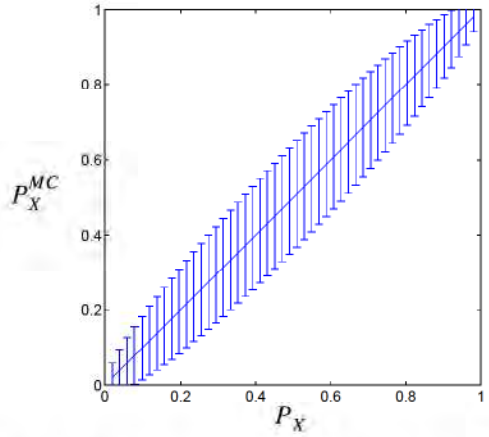


Figure 24.5 Expected Statistical Spread of P_X^{MC} for $N_T = 50$

By comparing Figure 24.4 with Figure 24.5, we can see that the spread in the data is explained by the statistical variations in P_X^{MC}. To further verify this observation, we generated additional testing data, in which 500 Monte Carlo trials were run to obtain each $P_X^{MC}(\mathbf{p})$ (i.e., $N_T = 500$). We applied this testing data to the network that was trained on the original data set with $N_T = 50$. The resulting scatter plots are shown in Figure 24.6. Here we can see that the spread has decreased dramatically from Figure 24.4, even though the network has not changed. This means that the neural network is fitting the true probabilities $P_X(\mathbf{p})$ and not the statistical fluctuations in $P_X^{MC}(\mathbf{p})$.

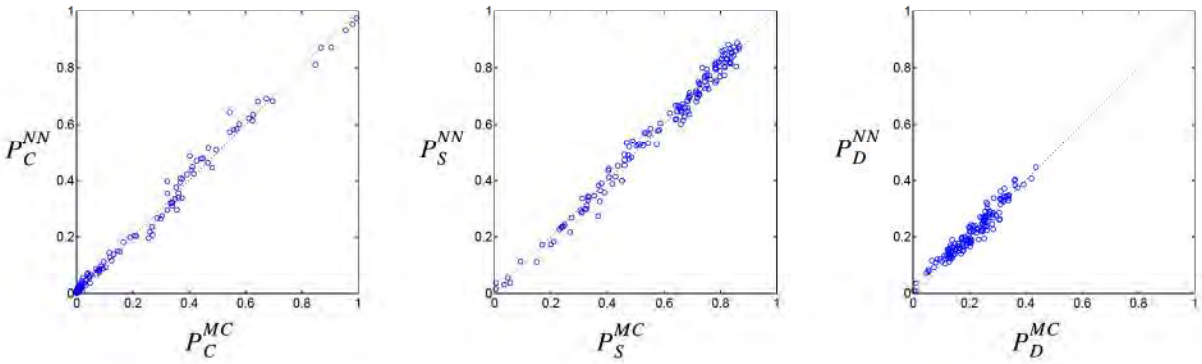


Figure 24.6 Scatter Plot of Network Outputs vs. Targets ($N_T = 500$)

After the neural network has been trained, it becomes a simple matter to investigate the effect of the input parameters on the reaction probabilities. In Figure 24.7, we see the effect of the impact parameter b on the reaction probabilities, as determined by the neural network. As the impact parameter is increased, the probability of chemisorption decreases, while the probabilities of scattering and desorption increase. (For this study, we have set θ to 5.4 radians, ϕ to 0.3 radians, v_{rot} to 0.004 radians per femtosecond, and v_{trans} to 0.004 angstroms per femtosecond.)

With standard methods, a study such as that shown in Figure 24.7 would take thousands of simulations. The trained neural network has fully captured the relationships between the parameters in $\mathbf{p}$ and the reaction probabilities. Therefore, we can perform arbitrary studies by simply computing the network responses at a varying set of input points. Note that the network interpolated smoothly through a noisy set of data points to capture the true underlying function. By using the early stopping technique, we prevented the network from overfitting the noise in the data.

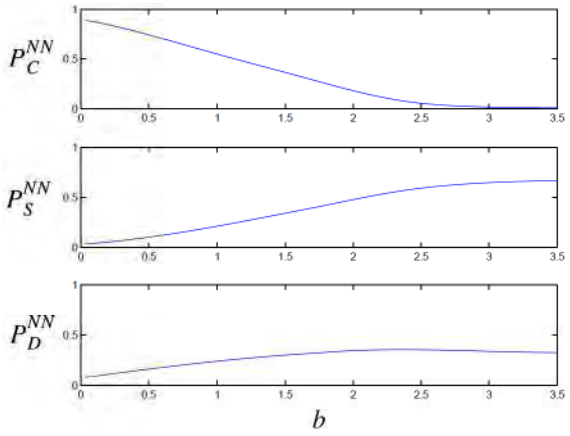


Figure 24.7 Reaction Probabilities vs. Impact Parameter

Data Sets

There are four data files associated with this case study:

- cvd_p.txt — contains the input vectors in the original data set
- cvd_t.txt — contains the target vectors in the original data set
- cvd_p500.txt — contains the input vectors in the $N_T = 500$ test set
- cvd_t500.txt — contains the target vectors in the $N_T = 500$ test set

They can be found with the demonstration software, which is described in Appendix C.

Epilogue

This chapter has illustrated the use of neural networks for probability estimation on a chemical vapor deposition problem. Monte Carlo simulations were used to provide estimates of the reaction probabilities. These estimates were used as targets for the neural network. The network was able to capture the true underlying probability function without overfitting to the errors in the Monte Carlo estimates. This was accomplished by using the early stopping procedure, which stops network training if the error on an independent validation set increases.

In the next chapter, we apply neural networks to a pattern recognition problem. We will also use multilayer neural networks for that application.

Further Reading

[AgSa05] P.M. Agrawal, A.N.A. Samadh, L.M. Raff, M. Hagan, S. T. Bukkapatnam, and R. Komanduri, "Prediction of molecular-dynamics simulation results using feedforward neural networks: Reaction of a C2 dimer with an activated diamond (100) surface," *The Journal of Chemical Physics* 123, 224711, 2005.

This paper describes the details of training a neural network to predict the reaction probabilities for chemical vapor deposition of diamond.

[Mill93] M.F. Miller, "A scaled conjugate gradient algorithm for fast supervised learning," Neural Networks, vol. 6, pp. 525-533, 1993.

The scaled conjugate gradient algorithm is a fast batch training algorithm for neural networks that requires a minimum of memory and computation at each iteration.

[RaMa05] L.M. Raff, M. Malshe, M. Hagan, D.I. Doughan, M.G. Rockley, and R. Komanduri, "*Ab initio* potential-energy surfaces for complex, multi-channel systems using modified novelty sampling and feedforward neural networks," The Journal of Chemical Physics, 122, 084104, 2005.

This paper explains how neural networks can be used for molecular dynamics simulations.

25 Case Study 3: Pattern Recognition

Objectives	25-1
Theory and Examples	25-2
Description of Myocardial Infarction Recognition	25-2
Data Collection and Preprocessing	25-3
Selecting the Architecture	25-6
Training the Network	25-7
Validation	25-7
Data Sets	25-10
Epilogue	25-11
Further Reading	25-12

Objectives

This chapter presents a case study in using neural networks for pattern recognition. In pattern recognition problems, you want a neural network to classify inputs into a set of target categories, e.g., recognize the vineyard that a particular bottle of wine came from, based on a chemical analysis, or classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis.

In this chapter we will demonstrate the application of multilayer neural networks to the recognition of heart disease from a reading of the electrocardiogram. We will show each of the steps in the pattern recognition process: data collection, feature extraction, architecture selection, network training and network validation.

Theory and Examples

In pattern recognition (pattern classification) problems we are trying to categorize network inputs into their corresponding classes. Here are a few examples of pattern recognition problems:

- recognition of handwritten zip codes
- spoken word recognition
- disease recognition from a list of symptoms
- fingerprint recognition
- white blood cell classification

In the case study presented in this chapter, we will be looking for patterns in electrocardiogram signals that indicate the presence of a myocardial infarction (heart attack).

Description of Myocardial Infarction Recognition

An electrocardiogram (EKG) is a recording of the electrical activity of the heart over time. It generally consists of an array of different signals recorded at the same time. An EKG can consist of a single signal (also called a *lead*), although the standard EKG that is used for detailed interpretation consists of 12 leads. EKG's with as many as 15 leads are sometimes used. Each lead represents the electrical activity across two points on the body. The 12-lead EKG is determined from 10 electrodes that are placed on specific locations on the body. The calculation of the 12-lead potentials from the 10 electrodes is a somewhat complex calculation, and beyond the scope of this case study. The interested reader is referred to [Dubi00] for a more complete discussion of the EKG.

Through a careful analysis of the EKG, a physician can often determine the health of the heart. The shapes of the signals indicate the path of electrical flow in the heart as various muscles are contracted in a coordinated way to pump blood in and out. If a part of the heart muscle has been damaged because of a lack of blood flow through the coronary arteries (called a myocardial infarction (MI), or heart attack), then the path of electrical flow changes. A well-trained physician can discern from the changes in the EKG, if the heart has been damaged, and where the damage has occurred.

For this case study, we will train a neural network to recognize MI's, using information obtained from a 15 lead EKG.

Data Collection and Preprocessing

The EKG signals used for this case study were obtained from the PhysioNet database [MoMa01]. Data were extracted from the QT data set for healthy patients and patients with MI's. Each EKG consists of 15 leads. The leads are labeled I, II, III, aVR, aVL, aVF, V1, V2, V3, V4, V5, V6, VX, VY, VZ. Figure 25.1 shows a small portion of the lead I signal for one of the healthy patients.

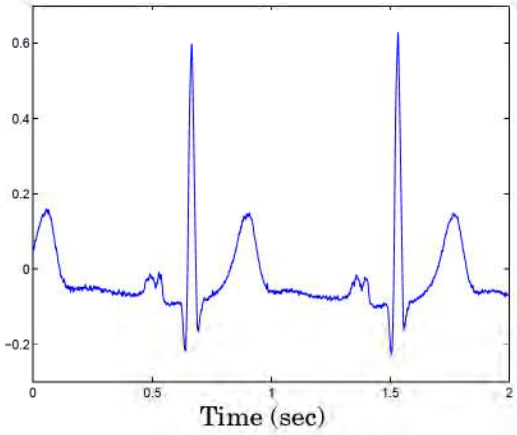


Figure 25.1 Example EKG Signal

Our data set has a total of 447 EKG records. Of these, 79 represent healthy patients, and the remaining 368 have an MI diagnosis. A diagnosis for each record was provided by a physician, but it is possible for the diagnosis to be in error. We will have more to say about this when we come to the validation of the network.

Each EKG consists of 15 leads measured at a rate of 1000 Hz for a period of several minutes. This is an enormous amount of data, and it would be impractical to use the entire EKG as an input to the neural network. As with many pattern recognition problems, we need to perform a *feature extraction* step before using the neural network to execute the pattern recognition step. Feature extraction involves mapping the high-dimensional input space into a space with fewer dimensions, in order to simplify and make more robust the pattern recognition step.

There are a number of general methods for dimensionality reduction. This includes linear methods, like the principal components method that we mentioned in Chapter 22, and nonlinear methods, like manifold learning [TeSi00]. For this case study, instead of using general methods to generate the low-dimensional feature space, we will extract features that are com-

monly used by physicians to detect abnormalities in the EKG. The first step is to consider a typical cycle of an EKG signal, as shown in Figure 25.2.

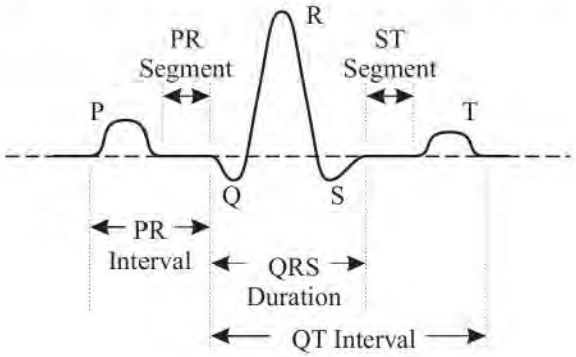


Figure 25.2 Prototype Cycle of an Electrocardiogram Signal

William Einthoven, in the early 1900's, was the first to carefully measure and analyze the EKG. He assigned the letters P, Q, R, S and T to the various deflections shown in the prototype cycle of Figure 25.2. He also described the electrocardiographic features of a number of cardiovascular disorders. He won the Nobel Prize in Medicine in 1924 for his discoveries. His features are still used to this day.

For this study, we have used some of the features that are in standard use by physicians, as well as other features that are related to the prototype cycle [Raff06]. The descriptions of the 47 features that we used are listed below. (If a description refers to the "amplitude" lead, this refers to the square root of the sum of squares of the VX, VY and VZ leads.)

Input Features to the Neural Network

1. age in years
2. gender, -1=female, 1=male
3. maximum heart rate in beats/min
4. minimum heart rate in beats/min
5. average time between heart beats in sec
6. rms deviation of the mean heart rate in beats/sec
7. full width at half maximum for the heart rate distribution
8. average qt interval for lead with max t wave
9. average qt interval for all leads
10. average corrected qt interval for lead with max t wave
11. average corrected qt interval for all leads
12. average qrs interval for all leads
13. average pr interval for lead with maximum p wave
14. rms deviation of pr intervals from average-max p lead

15. average pr interval for all leads
16. rms deviation for pr interval from average-all leads
17. percentage of negative p waves-max p lead
18. average percentage of negative p waves for all leads
19. maximum amplitude of any t wave
20. rms deviation of qt intervals
21. rms deviation of corrected qt intervals
22. average st segment length
23. rms deviation of st segment lengths
24. average heart rate in beats/min
25. rms deviation of heart rate distribution in beats/min
26. average rt angle averaged over all amplitude beats
27. number of missed r waves (beats)
28. % total qt intervals not analyzed or missing
29. % total pr intervals not analyzed or missing
30. % total st intervals not analyzed or missing
31. average number of maxima between t wave end and q
32. rms deviation of rt angle for all beats
33. ave qrs from amplitude lead
34. rms deviation of qrs from amplitude lead
35. ave st segment from amplitude lead
36. rms deviation of st segment from amplitude lead
37. ave qt interval from amplitude lead
38. rms deviation of qt interval from amplitude lead
39. ave bazetts corrected qt interval from amplitude lead
40. rms deviation of corrected qt interval from amplitude lead
41. ave r-r interval from amplitude lead
42. rms deviation of r-r interval from amplitude lead
43. average area under qrs complexes
44. average area under s-t wave end
45. average ratio of qrs area to s-t wave area
46. rms deviation of rt angle within each beat averaged over all beats in amplitude signal
47. st elevation at the start of the st interval for amplitude signal

To summarize, the data set contains 447 records. Each record has 47 input variables, and one target value. The target is 1 for a healthy diagnosis and -1 for an MI diagnosis.

One of the problems with the data set is that there are only 79 records for the healthy diagnosis, while there are 368 records for the MI diagnosis. If we train the network using the sum square error performance index, where all of the errors are weighted equally, the network will be biased to indicate

the MI diagnosis. The ideal solution to this problem would be to collect more data from healthy patients. Let's say that this is not possible in this case, and we need to do what we can with the data available. One possibility is to use a weighted sum square error as the performance index, where errors for healthy patients would be weighted higher than errors for MI patients, so that overall healthy and MI contributions would be equal if each error were equal. Another simple approach is to repeat the healthy records in the data set, so that the total number of healthy records is equal to the number of MI records. This requires extra computation, but that is not a problem in this case study. Since it is the simplest solution, we will use it here.

After the data has been collected, the next step is to divide the data into training, validation and test sets. In this case, we randomly set aside 15% of the data for validation and 15% for testing. For the validation and testing sets, we did not include multiple entries of the healthy records. This was done only for the training set.

The data were normalized using Eq. (22.1), so that the inputs were in the range [-1,1]. Since the tangent-sigmoid transfer function will be used in the output layer of the neural network, the targets were set to values of -0.76 and +0.76, instead of -1 and 1, to prevent training difficulties caused by saturation of the sigmoid functions, as discussed in Chapter 22.

Selecting the Architecture

Figure 25.3 shows the network architecture. We are using the tangent-sigmoid transfer function in both layers. This is the standard network for pattern recognition. There are cases in which two hidden layers are used, but we normally try first with one hidden layer. The number of neurons in the hidden layer, S^1, will depend on the complexity of the decision boundaries needed for the pattern recognition task. This is something that cannot generally be known before training. We will start with 10 neurons in the hidden layer, and then test the network performance after training.

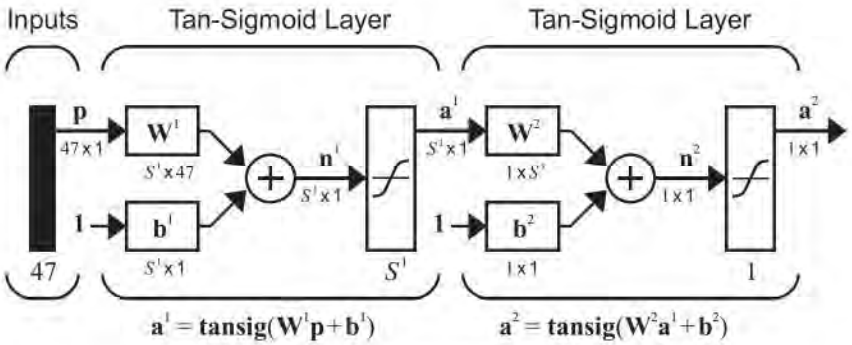


Figure 25.3 Network Architecture

Training the Network

We trained the network using the scaled conjugate gradient algorithm of [Mill93]. This algorithm is very efficient for pattern recognition problems. We used early stopping to prevent network overfitting.

Figure 25.4 illustrates the mean squared error versus iteration number. The blue line shows the validation error, and the black line shows the training error. We used a network with 10 neurons in the hidden layer ($S^1 = 10$). The minimum validation error occurred at iteration 16, as indicated by the circle in Figure 25.4, and the network parameters were saved at this point. Note that the validation error curve does not always fall at each iteration, and it may rise before falling to a lower value. We tested that the validation error was not reduced over 40 iterations before we finally stopped the training.

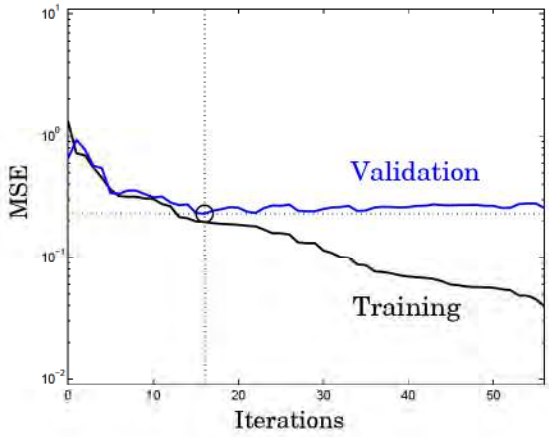


Figure 25.4 Mean Squared Error vs. Iteration Number ($S^1 = 10$)

Validation

As we discussed in previous chapters, an important tool for network validation in function approximation problems is a scatter plot of network outputs versus targets. For pattern recognition problems, the network outputs and targets are discrete variables, so a scatter plot is not particularly useful. Instead of the scatter plot, we use the confusion matrix, which was discussed in Chapter 22. Figure 25.5 shows the confusion matrix for our trained network on the test data. The upper left cell shows that 13 of the 14 healthy EKG's in the test set were classified correctly, while the 2,2 cell shows that 66 of the 71 MI EKG's were classified correctly. A total of 92.9% of the test data were classified correctly. The largest number of mistakes were for MI records that were classified as healthy (5), as shown in cell 1,2.

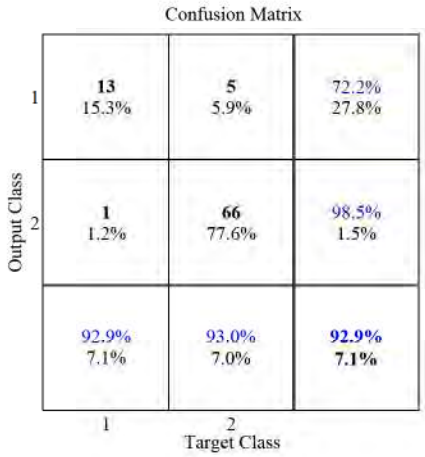


Figure 25.5 Confusion Matrix for Test Data (One Data Division)

Another useful validation tool for pattern recognition problems is the Receive Operating Characteristic (ROC) curve, described in Chapter 22. Figure 25.6 shows the ROC curve (blue line) for the test data. The ideal curve would follow the path from 0,0 to 0,1 and then to 1,1. The curve for this test set is close to the ideal path.

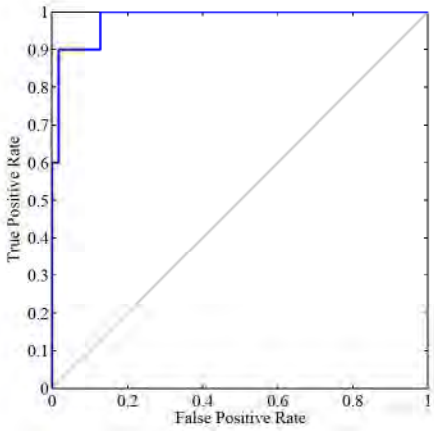


Figure 25.6 Receiver Operating Characteristic Curve (Test Set)

The results shown in Figure 25.5 and Figure 25.6 represent one division of the data into training/validation/testing sets. Because the data set is fairly small, especially in terms of healthy diagnoses, we might wonder how sensitive the results are to the data division. To investigate this sensitivity, we performed a Monte Carlo simulation. The data were divided 1,000 different

times. For each division of the data, a neural network was trained with different random initial weights. The results of the 1,000 trials were averaged together, and the results are shown in Figure 25.7.

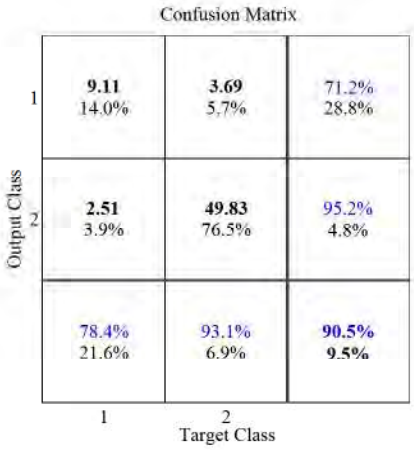


Figure 25.7 Average Test Confusion Matrix for 1,000 Monte Carlo Runs

Figure 25.7 represents the average results over the 1,000 different networks and data divisions. There were approximately 12 healthy patients (on average) in each test set. Of these, more than 9 were correctly diagnosed. There were approximately 54 sick patients in each test set, and approximately 50 were correctly diagnosed. The average testing error was approximately 9.5%. Note that none of the patients in the test set were used to train the neural network, so these numbers should be conservative estimates of how the network should perform on new patients.

The average test results for the Monte Carlo simulation are similar to our original test results. However, in addition to knowing the average results, it is also helpful to look at the distribution of errors. Figure 25.8 shows a histogram of the percentage errors. The average percent error is 9.5%, but there is a significant spread in the distribution of errors. The standard deviation of the percent error is 3.5.

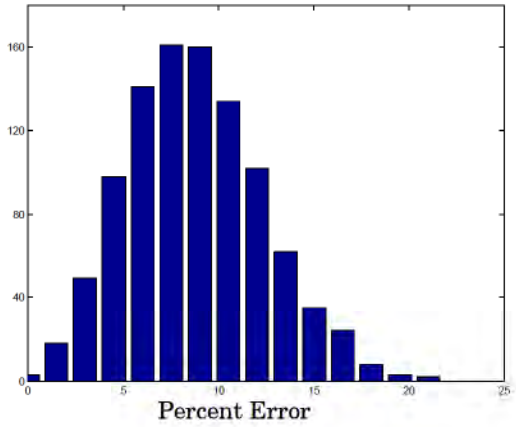


Figure 25.8 Percent Error Histogram (1000 Monte Carlo Trials)

This Monte Carlo process can be helpful in validating the data and the training process. For example, it is possible to identify which patients are misclassified in each Monte Carlo run. Those patients who are consistently misclassified (regardless of the division of the data) can be carefully investigated. These cases can be helpful in two areas. First, they can enable us to refine the data base. If, after reevaluation by clinicians, it is determined that a particular patient was mislabeled in the original data set, the data can be corrected. Second, if we find upon review that the patient was correctly labeled in the original data set, then we can use that patient to help improve the operation of the neural network classification. This may involve identifying new features which capture the key characteristics of the EKG, or it may involve obtaining more data with similar characteristics, in order to reinforce the training of the neural network.

The Monte Carlo process can be also used to help to improve the network performance. By combining the individual networks obtained from the Monte Carlo trials, we can often obtain a more accurate classification. The same input is applied to all of the networks, and the outputs can be combined through a "voting" procedure. We choose the class that is selected by the largest number of networks.

Data Sets

There are two data files associated with this case study:

- ekg_p.txt — contains the input vectors in the ekg data set
- ekg_t.txt — contains the targets in the ekg data set

They can be found with the demonstration software, which is described in Appendix C.

Epilogue

This chapter has demonstrated the use of multilayer neural networks for pattern recognition. In this case study, the pattern recognition network was used to classify EKG records into healthy and myocardial infarction diagnoses.

Most pattern recognition tasks involve a feature extraction step, in which the original data set is reduced in dimension. The features that were extracted from the EKG data consisted of characteristics of the prototype EKG cycle.

A Monte Carlo procedure was used as part of the network validation. The data were randomly divided a number of times into training/validation/test sets, and for each division a neural network was trained with random initial weights. The performances of all of the networks were analyzed to determine expected future performance. In addition, those records that were misclassified by most of the networks, regardless of the data division, were analyzed to assist in refining the data set and improving the pattern recognition.

In the next chapter, we apply neural networks to a clustering problem. We will use a self-organizing feature map network for that application.

Further Reading

[Dubi00] D. Dubin, *Rapid Interpretation of EKG's*, Sixth Edition, Tampa, FL: COVER, 2000.

This book describes the EKG in very clear terms, and leads you through the interpretation in a step-by-step way.

[MoMa01] G.B. Moody, R.G. Mark, and A.L. Goldberger, "PhysioNet: a Web-based resource for the study of physiologic signals," *IEEE Transactions on Engineering in Medicine and Biology*, vol. 20, no. 3, pp: 70-75, 2001.

This paper describes the PhysioNet data base that contains a large variety of recorded physiologic signals. The data base can be found at http://www.physionet.org/.

[Raff06] The features in the data set described in this chapter were designed and extracted by Dr. Lionel Raff, Regents Professor of Chemistry at Oklahoma State University.

[TeSi00] J. B. Tenenbaum, V. de Silva, J. C. Langford, "A Global Geometric Framework for Nonlinear Dimensionality Reduction," *Science*, vol. 290, pp. 2319-2323, 2000.

There are several different approaches to manifold learning, in which data is mapped from a high-dimensional space to a lower-dimensional manifold. This paper introduces a method called Isomap.

26 Case Study 4: Clustering

Objectives	26-1
Theory and Examples	26-2
Description of the Forest Cover Problem	26-2
Data Collection and Preprocessing	26-4
Selecting the Architecture	26-5
Training the Network	26-6
Validation	26-7
Data Sets	26-10
Epilogue	26-11
Further Reading	26-12

Objectives

This chapter presents a case study in using neural networks for clustering. In clustering problems, you want a neural network to group data by similarity. For example, market segmentation can be done by grouping people according to their buying patterns, data mining can be done by partitioning data into related subsets, and bioinformatic analysis can be done by grouping genes with related expression patterns.

In this chapter, we will apply clustering to a problem in forestry, in which we would like to analyze forest cover types. We will use the self-organizing feature map network of Chapter 16 to perform the clustering, and we will demonstrate a variety of visualization tools that can be used in conjunction with the SOFM.

Theory and Examples

This chapter presents a case study in using neural networks for clustering. In clustering problems, we generally don't have a set of network targets available, so clustering networks are trained by unsupervised training algorithms. Instead of training a network to produce a desired response, we want to analyze a data set to look for hidden patterns. There are many application areas for clustering. It is widely used in data mining, in which we analyze large data sets to identify similarities within subsets of the data. It is used in city planning, when town councils apportion regions of the city into areas of similar home type and land usage. It is used in image compression, in which a small set of prototype sub-images are identified and combined to represent a large collection of images. It is used in speech recognition systems, in which speakers are clustered into categories in order to simplify the problem of speaker-independent recognition. Clustering is used by marketers to identify distinct groups in their customer bases. It has also been used to organize large bibliographic data bases so that related material can be quickly accessed.

The neural network that we will use in this application is the self-organizing feature map (SOFM), which we introduced in Chapter 16. This clustering network has a unique attribute that enables us to visualize large data sets in many dimensions. We will focus on that visualization capability in this case study.

Description of the Forest Cover Problem

An important job of the forest service is to maintain accurate natural resource inventory information. One key characteristic that is recorded is the type of forest cover found in wilderness areas. This type of data can be expensive to collect, since it generally requires on-site inspection or estimation from remotely sensed data. [BlDe99] describes how forest cover type can be predicted from independent variables that can be more easily obtained. In this chapter, we will use the data described in that paper to perform a clustering analysis. We will demonstrate how an analysis of the data using an SOFM can allow us to visualize the high-dimensional space of independent variables and identify relationships between the forest cover types.

Ten independent variables that can indicate forest cover type are shown in Table 26.1. (There were 12 variables used in [BlDe99], but for ease of presentation, we selected only the first 10 for this case study.) These variables can be measured or estimated much more easily than forest cover type. We want to use the SOFM to find out whether these variables can be used to cluster the data in such a way as to separate regions with different forest cover types.

Variable Number	Description	Units
1	Elevation in meters	meters
2	Aspect in degrees azimuth	azimuth
3	Slope in degrees	degrees
4	Horz Dist to nearest surface water	meters
5	Vert Dist to nearest surface water	meters
6	Horz Dist to nearest roadway	meters
7	Hillshade index at 9am, summer solstice	0 to 255 index
8	Hillshade index at noon, summer solstice	0 to 255 index
9	Hillshade index at 3pm, summer solstice	0 to 255 index
10	Horz Dist to nearest wildfire ignition points	meters

Table 26.1 Description of Independent Variables

The forest cover types of interest in [BlDe99] are shown in Table 26.1. The data set we will use for this case study contains information on cover type, but we will not use this as part of the training process. We will use it to test the clustering ability of the SOM.

Label	Name
0	Krummholz
1	Spruce/Fir
2	Lodgepole Pine
3	Ponderosa Pine
4	Cottonwood/Willow
5	Aspen
6	Douglas-fir

Table 26.2 Forest Cover Types

Data Collection and Preprocessing

The data used in this study came originally from [HeBa99]. It contains the forest cover type for 30 x 30 meter cells obtained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. The original data set contained 581,012 observations of 12 independent variables and the forest cover type. We have used the first 20,000 observations, and we have used only the first 10 independent variables, which are described in Table 26.1. The forest cover types are given in Table 26.2. As mentioned previously, we did not use these for training.

For supervised learning, as demonstrated in the previous three chapters, after the data is collected, the next step is to divide the data into training, validation and test sets. For unsupervised learning, we don't generally divide the data in this way, because there is no need for a validation set to stop the training. Competitive training is typically performed for a fixed number of iterations. We use the entire data set for training.

The next step is to normalize the data. The data were scaled using Eq. (22.1), so that the inputs were in the range [-1,1]. (For the SOFM, data can also be scaled using Eq. (22.2), so that the input variables have a mean of 0 and a variance of 1.)

Before proceeding to train the network, it is often useful to view the input data. One convenient format for this is the scatter plot. Figure 26.1 illustrates a set of scatter plots among the input variables 7, 8 and 9. The diagonal plots in this figure are histograms for these three input variables, and the off-diagonal plots are the scatter plots. (We only show three of the variables in this figure because of the limits of the page size.)

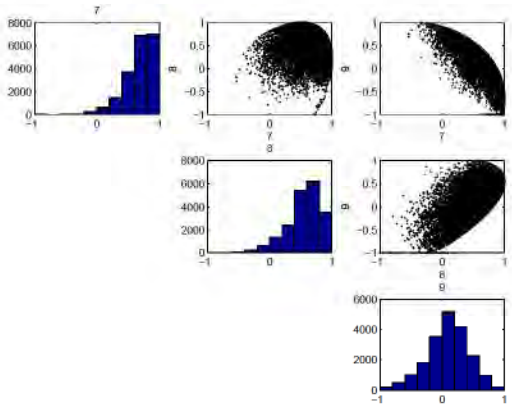


Figure 26.1 Scatter Plots for Input Variables 7, 8, 9.

There are several things we can look for in Figure 26.1. First, we want to see how well the data is scattered throughout the range. If some variables show little or no variation, then we would remove them from the analysis. We also look for correlation between the variables. For example, if the points in the scatter plot fell exactly along a line, then we would know that the two variables were linearly related. There would be no need to use both variables in the analysis. From Figure 26.1 we can see that there is some correlation between the variables, but they are not linearly dependent.

Selecting the Architecture

We will use the SOFM network, described in Chapter 16, to perform the clustering for this case study. The specific architecture is often selected based, in part, on the number of data points, so that there will be a reasonable amount of data associated with each prototype vector. (Recall that each row of the weight matrix represents a prototype vector. An input is associated with the prototype vector to which it is closest.) As the data set size increases, the number of neurons should increase as well, although not as rapidly. A rule of thumb is to have the number of neurons increase as the square root of the number of data points.

Figure 26.2 shows the architecture of the network we selected. We have 10 input variables (defined in Table 26.1), and we are using 150 neurons. The feature map is 15x10, and it uses an hexagonal arrangement of neurons. This means that each internal neuron will have six neighbors.

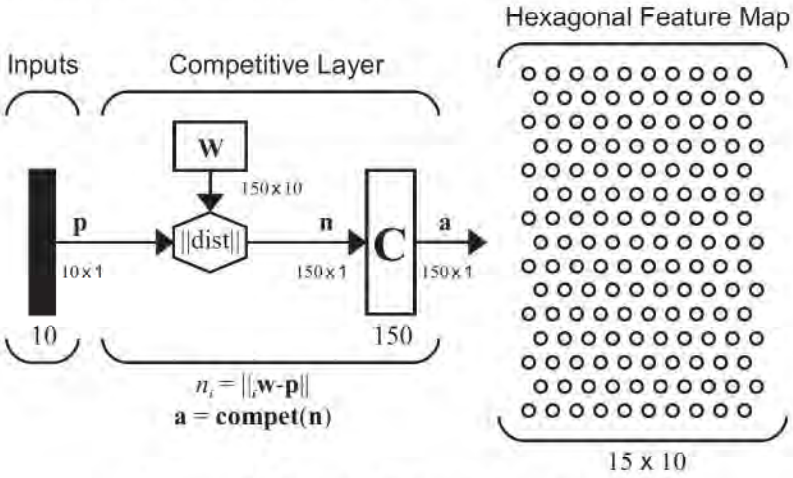


Figure 26.2 SOM Network Architecture

After the network has been trained, we will analyze the results to determine if the network architecture is satisfactory. In practical cases, we often try several different architectures. Unlike with supervised training, in which we have a clear performance measure (normally sum square error), there is no firm criterion for best performance in SOFM networks. Often

what we are looking for is insight into the data set. It is somewhat of an art to selecting the best architecture and training regime for SOFM networks. This will become more clear when we analyze the results of the trained network in a later section.

Training the Network

Before beginning the training, the weight vectors (rows of **W**) were initialized using what is called *linear initialization* [Koho95]. First, a covariance matrix of the input vectors was computed. Then, the two eigenvectors of this matrix having the two largest eigenvalues were found. The rows of **W** were then assigned by taking the average of the input vectors and adding linear combinations of the two eigenvectors. This places all of the initial weight vectors in the space spanned by the two eigenvectors. This initialization process produces quicker training convergence than when using a purely random weight initialization. (It is also possible to randomly select input vectors from the training set to be the initial weight vectors.)

Recall from Eq. (16.21) the SOFM learning rule, which we repeat here:

$$\begin{aligned} {}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\ &= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \end{aligned} \qquad i \in N_{i^*}(d) \tag{26.1}$$

where i^* is the index of the winning neuron, and

$$N_i(d) = \{j, d_{ij} \le d\} \tag{26.2}$$

defines the neuron neighborhood. For this case study, we have used a batch form of the algorithm, in which all of the inputs in the training set are applied to the network before the weights are updated. To develop the batch form, we can first modify the sequential form of Eq. (26.1) to

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + h_{i^*,i}(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)), \tag{26.3}$$

where $h_{i^*,i}$ is the neighborhood function. The neighborhood function that would produce Eq. (26.1) is

$$h_{i^*,i} = \begin{cases} \alpha & i \in N_{i^*}(d) \\ 0 & i \notin N_{i^*}(d) \end{cases} \tag{26.4}$$

Using this definition of neighborhood function, we can define a batch version of Eq. (26.1):

$${}_i\mathbf{w}(k) = \frac{\sum_{q=1}^{Q} h_{i^*(q),i}\mathbf{p}(q)}{\sum_{q=1}^{Q} h_{i^*(q),i}}, \tag{26.5}$$

where k is the iteration number and $i^*(q)$ is the winning neuron for input $\mathbf{p}(q)$. Note that for the batch algorithm we have to distinguish between the iteration number and the input number, since all inputs are applied to the network at each iteration. This is in contrast to the sequential algorithm of Eq. (26.1), where there is one iteration for each input. Also, notice that the learning rate does not affect the batch algorithm, since it would appear in both the numerator and the denominator of Eq. (26.5).

For the neighborhood function of Eq. (26.4), this batch algorithm has the effect of assigning each weight to the average of the input vectors for which it is in the neighborhood of the winner. As with the sequential algorithm, the neighborhood size is decreased during training. The neighborhood size is set large at the beginning of training until all weights move into the region of the input space where the data lies. Then the neighborhood size is reduced, to fine-tune the position of the weights.

The batch algorithm requires many fewer iterations than the sequential algorithm, although each iteration requires much more computation. For this case study, we used two iterations of the batch algorithm. During the first iteration the neighborhood size was 4, and during the second iteration the neighborhood size was reduced to 1.

Validation

We will consider two numerical measures of the quality of a trained SOM: resolution and topology preservation (see page 22-23). One measure of SOM resolution is the *quantization error*, which is the average distance between each data vector and its winning neuron. If the average distance is too large, then there are many input vectors that are not adequately represented by any of the prototypes.

A measure of SOM topology preservation is the *topographic error*. This is the proportion of all input vectors for which the closest (winning) neuron and the next closest neuron are not adjacent to each other in the feature map topology. When this number is small, it means that the neurons that are neighbors in the topology are also neighbors in the input space. It is important that this topology be preserved, so that the visualization tools we will discuss later can provide valid insight into the data set.

For our trained SOM, the final quantization error was 0.535, and the final topographic error was 0.037. This means that for less than 4% of all input vectors, the winning neuron and the next closest neuron were not adjacent to each other. It appears that the SOM has achieved the correct topology by the completion of the training.

There are a number of visualization methods that can be used to assess the trained SOM network. One of the key tools is called the unified distance matrix, or *u-matrix*. This is a figure that shows the distance between neighboring neurons in the feature map. The u-matrix has a cell for each neuron

in the feature map and an additional cell between each pair of neurons. The cells between neurons are color-coded with the distance between the corresponding weight vectors. The cells that represent the neurons are coded with the mean of the surrounding values. Figure 26.3 shows the u-matrix for our trained SOM.

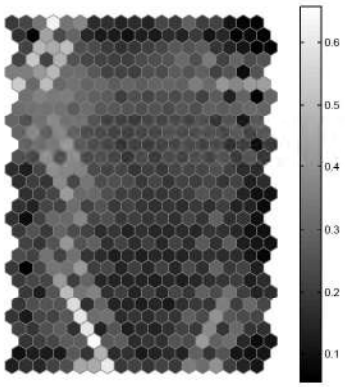


Figure 26.3 U-Matrix for Trained SOM

In Figure 26.3, the light-colored cells represent large distances between neurons. We can see that there is a string of light colored cells on the left side of the feature map. This indicates that the clusters associated with the neurons on the left side of the map are significantly different than those in the middle and right sides of the map. For this data set we actually know the forest cover types for each data point. We can label the feature map cells with the cover type that is associated with the closest input vector to that cluster center. The resulting labeled map is shown in Figure 26.4.

By comparing Figure 26.4 with Figure 26.3, we can see that the forest cover type 2 (see Table 26.2) is associated with the left edge of the feature map. As we move from left to right across the map, we see type 0 and 1 coded into the center section, followed by types 5, 3 and 4, with type 6 located mainly in the upper right section of the map. It is clear that the SOM has learned to cluster the data according to forest cover type.

To get more insight into how the SOM has clustered the data, we can produce a "hit histogram." For this graph, we count how many times each neuron was the winning neuron for the entire data set. Since our data is labeled with forest cover type, we can also see where each type falls on the feature map. Such a graph is displayed in Figure 26.5. In each cell you can see a hexagram with a certain gray-scale. The size of the hexagrams indicate how many times the corresponding neuron was the winning neuron. The gray level of the hexagram indicates the forest cover type. The darkest hexagons correspond to type 0 forest covers, and the lightest hexagons correspond to type 6 forest covers. We can see that the various regions of the

map have consistent colors. The left side has medium gray levels, corresponding to type 2 cover. The darkest levels are in the center-left region of the map, which corresponds to cover types 0 and 1. The lighter levels, which correspond to types 5 and 6, are in the center-right region, and the median levels of gray, corresponding to types 3 and 4, are on the right edge.

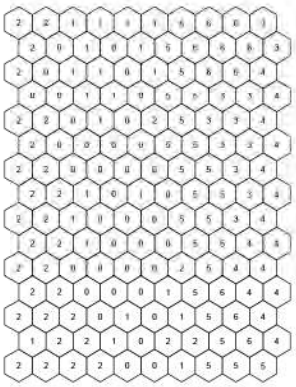


Figure 26.4 Labeled SOM

For many problems, we would not be able to label each input vector. The point here is that the SOM has been able to cluster the data into similar cover types, without knowing what the actual cover types were. This means that the 10 variables making up the input vectors have enough correlation with cover type to allow the SOM to make a useful clustering of the data.

Another tool that is useful in analyzing the trained SOM is the component plane. A component plane is a figure that represents a column of the weight matrix of the SOM. Each column corresponds to one element of the input vector; the jth element of column i represents the connection from input i to neuron j. In a component plane, each element of the weight is represented by a cell in the feature map topology at the location of the neuron to which it is connected. The gray level of the cell represents the magnitude of that element of the weight vector.

The ten component planes (one for each column of the weight matrix - each element of the input vector) for the trained SOM are shown in Figure 26.6. The first thing that we notice is that each of the columns is distinct. There are no two columns that have the same pattern. We can also see that input variables 1, 4, 5, 6 and 10 seem to be important in separating type 2 cover types from the rest of the data. They show patterns in which a boundary appears on the left edge of the feature map, where the type 2 cover types are clustered. By going back to Table 26.1, we can then locate the appropriate variables to see if we can deduce their connection to type 2 cover.

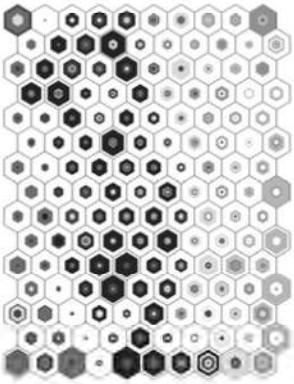


Figure 26.5 Hit Histogram for Trained SOM

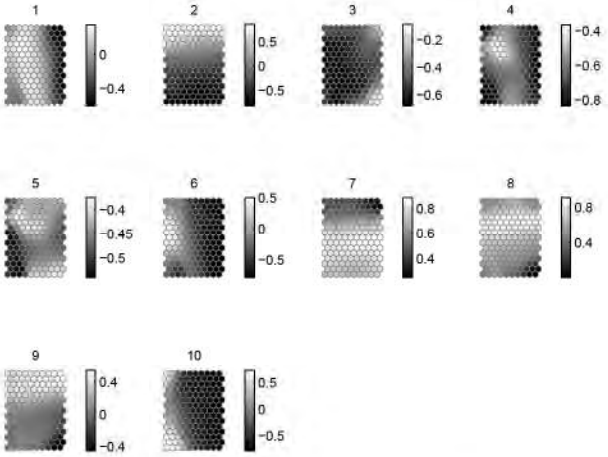


Figure 26.6 Component Planes for the Trained SOM

Data Sets

There are two data files associated with this case study:

- cover_p.txt — contains the input vectors in the data set
- cover_t.txt — contains the targets (labels) in the data set

They can be found with the demonstration software, which is described in Appendix C.

Epilogue

This chapter has demonstrated the use of SOM networks for clustering, in which input vectors in a data set are arranged so that similar vectors are placed in the same cluster. In this case study, the SOM was used to cluster forestry data. The idea was to cluster land into similar forest cover types.

One of the principal advantages of the SOM network, in addition to its ability to efficiently cluster a data set, is its ability to enable visualization of high dimensional data sets.

In the next chapter, we apply neural networks to a prediction problem. We will use a Nonlinear Autoregressive model with eXogenous inputs (NARX) network for that application.

Further Reading

[BlDe99] J. A. Blackard and D. J. Dean, "Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types from Cartographic Variables," Computers and Electronics in Agriculture, vol. 24, pp. 131-151, 1999.

This study compared neural networks and discriminant analysis for predicting forest cover types from cartographic variables. The study evaluated four wilderness areas in the Roosevelt National Forest, located in the Front Range of northern Colorado.

[HeBa99] S. Hettich and S. D. Bay, *The UCI KDD Archive* [http://kdd.ics.uci.edu], Irvine, CA: University of California, Department of Information and Computer Science, 1999.

The UCI Knowledge Discovery in Databases Archive. This is an online repository of large data sets which encompasses a wide variety of data types, analysis tasks, and application areas. It is maintained by the University of California, Irvine.

[Koho93] T. Kohonen, "Things you haven't heard about the Self-Organizing Map," *Proceedings of the International Conference on Neural Networks* (ICNN), San Francisco, pp. 1147-1156, 1993.

This paper describes the batch form of the SOM learning rule, as well as other variations on the SOM.

[Koho95] T. Kohonen, Self-Organizing Map, 2nd ed., Springer-Verlag, Berlin, 1995.

This text describes the theory and practical operation of the Self-Organizing Map in detail. It also has a chapter on the Learning Vector Quantization algorithms.

27 Case Study 5: Prediction

Objectives	27-1
Theory and Examples	27-2
Description of the Magnetic Levitation System	27-2
Data Collection and Preprocessing	27-3
Selecting the Architecture	27-4
Training the Network	27-7
Validation	27-8
Data Sets	27-13
Epilogue	27-14
Further Reading	27-15

Objectives

This chapter presents a case study in using neural networks for prediction. Prediction is a kind of dynamic filtering, in which past values of one or more time series are used to predict future values. Dynamic networks, such as those described in Chapter 10 and Chapter 14, are used for filtering and prediction. Unlike the previous case studies, the input to these dynamic networks is a time sequence.

There are many applications for prediction. For example, a financial analyst might want to predict the future value of a stock, bond, or other financial instrument. An engineer might want to predict the impending failure of a jet engine. Predictive models are also used for system identification (or dynamic modeling), in which we build dynamic models of physical systems. These dynamic models are important for analysis, simulation, monitoring and control of a variety of systems, including manufacturing systems, chemical processes, robotics and aerospace systems. In this chapter we will demonstrate the development of predictive models for a magnetic levitation system.

Theory and Examples

This chapter presents a case study in using neural networks for prediction. In this case study, the predictor neural network is used to model a dynamic system. This dynamic modeling from data is referred to as system identification. System identification can be applied to a variety of systems: economic, aerospace, biological, transportation, communication, manufacturing, chemical process, etc. For this case study, we will consider a simple magnetic levitation system. Magnetic levitation has been used for many years in transportation systems. In our simple maglev system, we will suspend a magnet above an electromagnet. A maglev train works on a similar principle.

Description of the Magnetic Levitation System

The objective of this maglev system is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as shown in Figure 27.1.

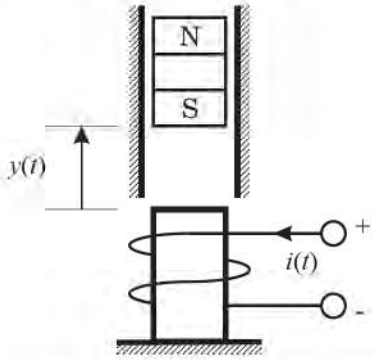


Figure 27.1 Magnetic Levitation System

The equation of motion for this system is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)\,\mathrm{sgn}[i(t)]}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt} \tag{27.1}$$

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet. For our case study, the parameter values are set to $\beta = 12$, $\alpha = 15$, $g = 9.8$, $M = 3$.

The objective of the case study will be to develop a dynamic neural network model that can predict the next value of the magnet position, based on previous values of the magnet position and the input current. Once the model has been developed, it can be used to find a controller that can determine the correct current to apply to the electromagnet, so as to move the magnet to some desired position. We won't go in to the control design in this case study, but the reader is referred to [HaDe02] and [NaMu97].

Data Collection and Preprocessing

For this case study, we did not build the maglev system of Figure 27.1. Instead, we created a computer simulation to implement Eq. (27.1). We used Simulink® to implement the simulation, but any simulation tool could be used. For our simulations, the current was allowed to range from -1 to 4 amps. The data were collected every 0.01 seconds.

To develop an accurate model, we need to be sure that the system inputs and outputs cover the operating range for which the system will be used. For system identification problems, we often collect training data while applying random inputs which consist of a series of pulses of random amplitude and duration. (This form of input is sometimes called a skyline function, because of its resemblance to a city skyline.) The duration and amplitude of the pulses must be chosen carefully to produce accurate identification. Figure 27.2 shows the input current, and the corresponding magnet position for our data set. A total of 4000 data points were collected.

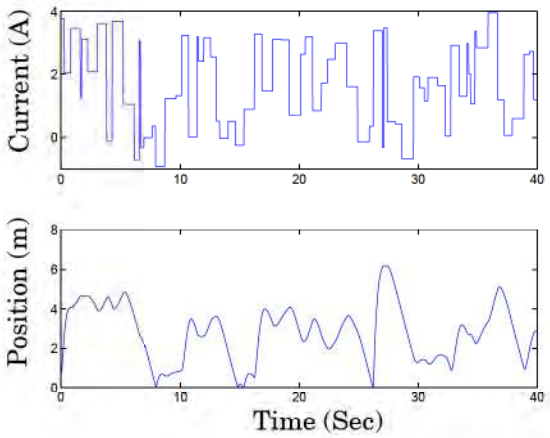


Figure 27.2 Magnetic Levitation Data

The skyline form of input function has the advantage that it can explore both transient and steady state operation of the system. Because some of the pulses are long, the system will approach steady state at the end of

those pulses. Shorter width pulses explore transient operation of the system.

When steady state performance is poor, it is useful to increase the duration of the input pulses. Unfortunately, within a training data set, if we have too much data in steady state conditions, the training data may not be representative of typical plant behavior. This is due to the fact that the input and output signals do not adequately cover the region that is going to be controlled. This will result in poor transient performance. We need to choose the training data so that we produce adequate transient and steady state performance. This can be done by using an input sequence with a range of pulse widths and amplitudes.

After the data has been collected, the next step is to divide the data into training, validation and test sets. In this case, because we will be using the Bayesian regularization training technique, we do not need to have a validation set. We did set aside 15% of the data for testing purposes. When the input is a time sequence, it is useful to have the testing sequence consist of a contiguous segment of the original data set. For our tests, we used the last 15% of the data as the testing set.

The data were scaled using Eq. (22.1), so that both the inputs and the targets were in the range [-1,1]. The resulting scaled data is shown in Figure 27.3

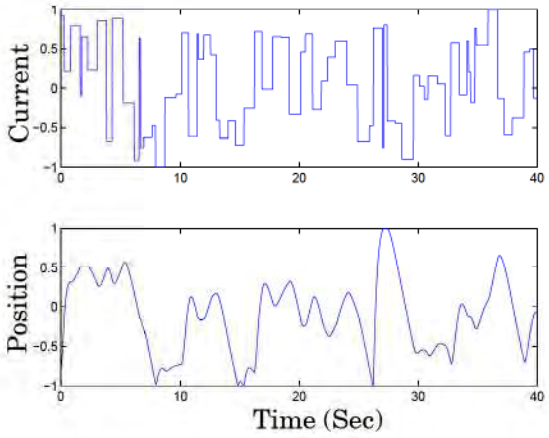


Figure 27.3 Scaled Data

Selecting the Architecture

There are many dynamic network architectures that can be used for prediction. A popular architecture is the nonlinear autoregressive network

with exogenous inputs (NARX) network, which was discussed in Chapter 22. The NARX network is a recurrent dynamic network, with feedback connections enclosing several static layers of the network. The NARX model is based on the linear ARX model, which is commonly used in time series modeling.

The defining equation for the NARX model is

$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u)), \quad (27.2)$$

where the next value of the dependent output signal $y(t)$ is regressed on previous values of the output signal and previous values of an independent (exogenous) input signal $u(t)$. (For our application, $y(t)$ is the position of the magnet, and $u(t)$ is the current going into the electromagnet.) We can implement the NARX model by using a feedforward neural network to approximate the function $f(\ )$. A diagram of the resulting network is shown in the Figure 27.4, where a two-layer feedforward network is used for the approximation. The output of the last layer of the network is the prediction of the next value of the magnet position. The network input is the current into the electromagnet.

We are using the tan-sigmoid transfer function in the hidden layer, and a linear output layer. As with the standard multilayer network, the number of neurons in the hidden layer, S^1, will depend on the complexity of the system being approximated. We will discuss this choice in the next section.

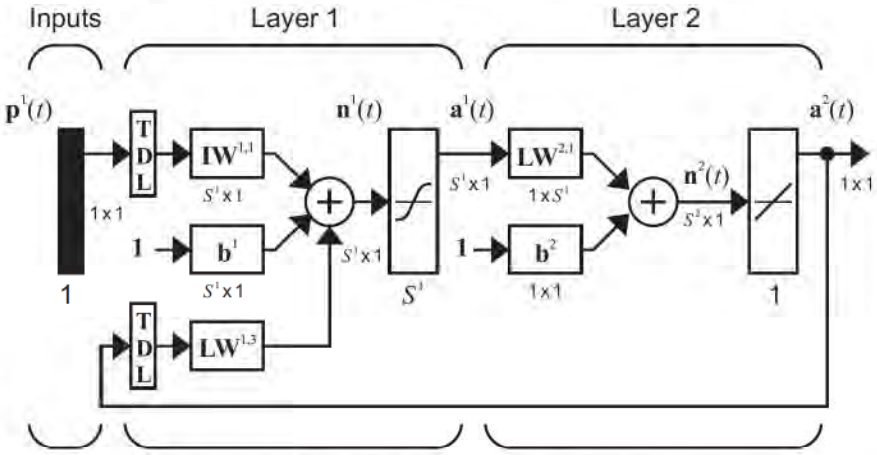


Figure 27.4 NARX Network Architecture

To define the architecture, we also need to set the length of the tapped-delay lines. The TDL for the inputs will contain the variables $u(t-1), \ldots, u(t-n_u)$, and the TDL for the outputs will contain the variables $y(t-1), \ldots, y(t-n_y)$. The TDL lengths n_u and n_y need to be defined. Be-

cause the defining differential equation in Eq. (27.1) is second order, we will start with $n_y = n_u = 2$. Later, we will investigate other possibilities.

Before demonstrating the training of the NARX network, we need to present an important configuration that is useful in training. We can consider the output of the NARX network to be an estimate of the output of the nonlinear dynamic system that we are trying to model. The output is fed back to the input of the feedforward neural network, as part of the standard NARX architecture, as shown on the left side of Figure 27.5. Since the true output is available during the training of the network, we could create a series-parallel architecture (see [NaPa90]), in which the true output is used instead of feeding back the estimated output, as shown on the right side of Figure 27.5. This has two advantages. The first is that the input to the feedforward network will be more accurate. The second is that the resulting network has a purely feedforward architecture, and static backpropagation can be used for training.

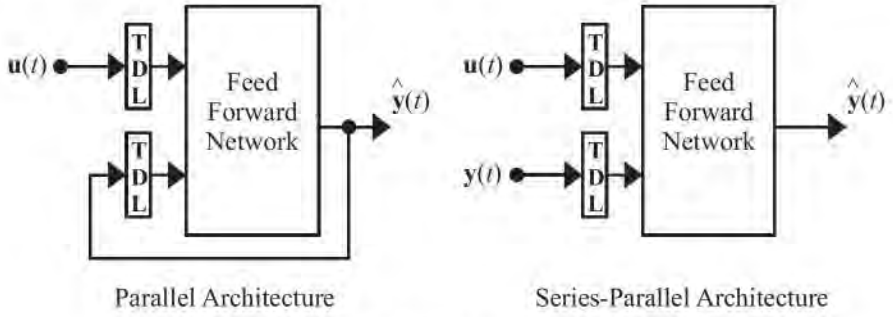


Figure 27.5 Parallel and Series-Parallel Forms

Using the series-parallel form, we can actually use a standard multilayer network to implement the NARX model. We can create an input vector that consists of previous system inputs and outputs:

$$\mathbf{p} = \begin{bmatrix} u(t-1) \\ u(t-2) \\ y(t-1) \\ y(t-2) \end{bmatrix}. \tag{27.3}$$

The target is then the next value of the output:

$$\mathbf{t} = \begin{bmatrix} y(t) \end{bmatrix}. \tag{27.4}$$

Training the Network

We used the Bayesian regularization training algorithm, described in Chapter 13, to train the NARX network, after the weights were initialized using the Widrow/Nguyen method (see page 22-13). The prediction problem is similar to the function approximation problem, which was demonstrated in Chapter 23, and the Bayesian regularization method is effective for both applications.

Because we have 4000 data points, and the number of network weights and biases will be less than 100 (as we will see later), the chances of overfitting are very small. We do not need to use Bayesian regularization (or early stopping) in this case. However, because it can tell us the effective number of parameters, we like to use it whenever it is appropriate.

Figure 27.6 illustrates the sum square error versus iteration number, while using Bayesian regularization. We used a network with 10 neurons in the hidden layer ($S^1 = 10$) for this case. The network was trained for 1000 iterations, at which time the performance was changing very little. Several different networks were trained with different initial conditions, and the final SSE was similar for each, so we can be confident that we did not reach a local minimum.

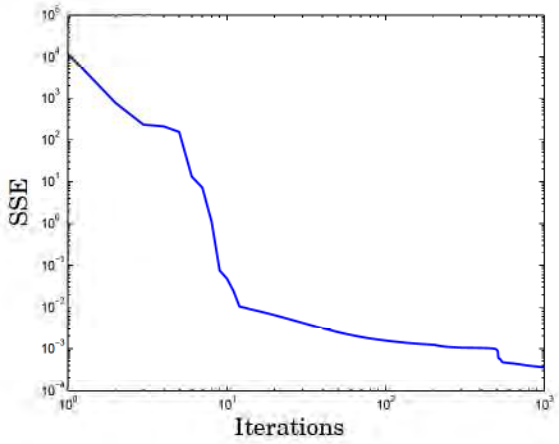


Figure 27.6 Sum Squared Error vs. Iteration Number ($S^1 = 10$)

In Figure 27.7, we can see the variation of the effective number of parameters γ during training. It eventually converges to 39. There are a total of 61 parameters in this 4-10-1 network, so we are effectively using less than 2/3 of the weights and biases. If the effective number of parameters was close to the total number of parameters, then we would increase the number of hidden neurons and retrain the network. That is not the case here.

There is no need to decrease the number of neurons, since the network computation time is not critical for this application. The only other reason for reducing the number of neurons would be to prevent overfitting. In terms of preventing overfitting, having 39 effective parameters is equivalent to having 39 total parameters. That is the beauty of using the Bayesian regularization technique. This method chooses the correct number of parameters for each problem, as long as we have a sufficient number of potential parameters in the network.

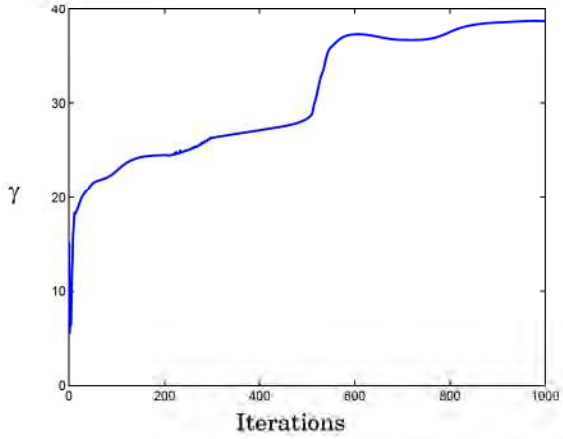


Figure 27.7 Effective Number of Parameters ($S^1 = 10$)

Validation

As we discussed in previous chapters, an important tool for network validation is a scatter plot of network outputs versus targets, as shown in Figure 27.8 (in normalized units). The figure on the left shows the training data, while the figure on the right shows the testing data. Because the testing data fit is as good as the training data fit, we can be confident that the network did not overfit.

For prediction problems, there is another set of tools for model validation. These tools are based on two basic properties of accurate prediction models. The first property is that the prediction errors,

$$e(t) = y(t) - \hat{y}(t) = y(t) - a^2(t), \tag{27.5}$$

should be uncorrelated with each other from one time step to the next. The second property is that prediction errors should be uncorrelated with the input sequence $u(t)$. (See [BoJe86].)

If the prediction errors were correlated with each other, then we could use that correlation to improve the predictions. For example, if prediction errors one time step apart had a positive correlation, then a large positive prediction error at the current time point would suggest that the prediction error at the next time point would also be positive. By lowering our next prediction, we could then reduce the next prediction error.

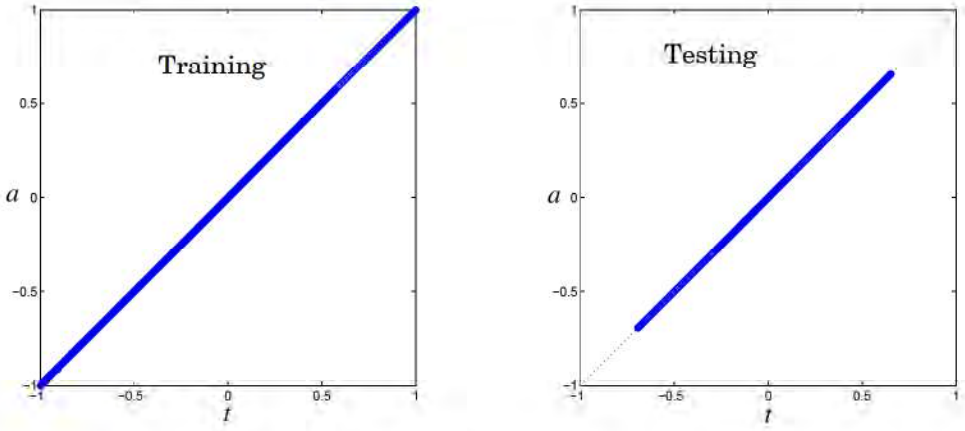


Figure 27.8 Scatter Plots of Network Outputs vs. Targets - Training and Testing Sets

The same argument holds for correlation between the input sequence and the prediction error. For accurate prediction models, there should be no correlation between the input and the prediction error. If there was correlation, then we could use this correlation to improve the predictor.

To measure the correlation in a time sequence, we use the autocorrelation function, which can be estimated by

$$R_e(\tau) = \frac{1}{Q-\tau} \sum_{t=1}^{Q-\tau} e(t)e(t+\tau) . \qquad (27.6)$$

The autocorrelation function of the prediction error (in normalized units) of our trained network for the maglev problem is shown in Figure 27.9.

For the prediction error to be uncorrelated (termed "white" noise), the autocorrelation function should be an impulse at $\tau = 0$, with all other values equal to zero. Because Eq. (27.6) provides only an estimate of the true autocorrelation function, the values at $\tau \neq 0$ will never be exactly equal to zero. If the error sequence is white noise, then we can find confidence intervals for the $R_e(\tau)$ (see [BoJe86]) defined by

$$\pm 2\frac{R_e(0)}{\sqrt{Q}}. \tag{27.7}$$

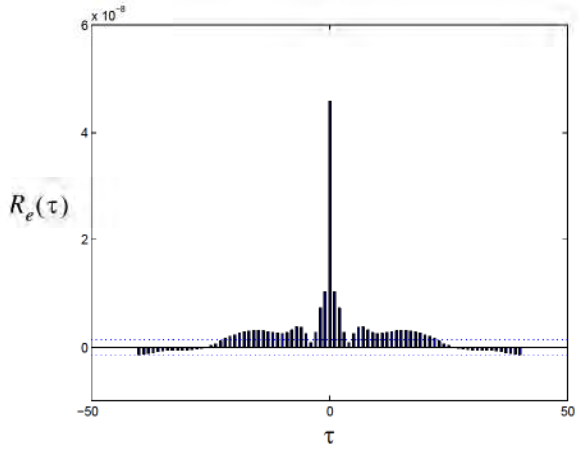


Figure 27.9 $R_e(\tau)$ $(n_y = n_u = 2, S^1 = 10)$

The dashed blue lines in Figure 27.9 indicate these confidence bounds. We can see that the estimated autocorrelation function for the prediction errors falls outside these bounds at a number of points. This indicates that we may need to increase n_y and n_u.

To measure correlation between the input sequence $u(t)$ and the prediction error $e(t)$, we use the cross-correlation function, which can be estimated by

$$R_{ue}(\tau) = \frac{1}{Q-\tau}\sum_{t=1}^{Q-\tau} u(t)e(t+\tau). \tag{27.8}$$

The cross-correlation function $R_{ue}(\tau)$ (in normalized units) of our trained network for the maglev problem is shown in Figure 27.9.

As with the estimated autocorrelation function, we can define confidence intervals to determine if the cross-correlation function is near zero

$$\pm 2\frac{\sqrt{R_e(0)}\sqrt{R_u(0)}}{\sqrt{Q}}. \tag{27.9}$$

The dashed blue lines in Figure 27.10 represent these confidence bounds. The cross-correlation function remains within these bounds, so it does not indicate a problem.

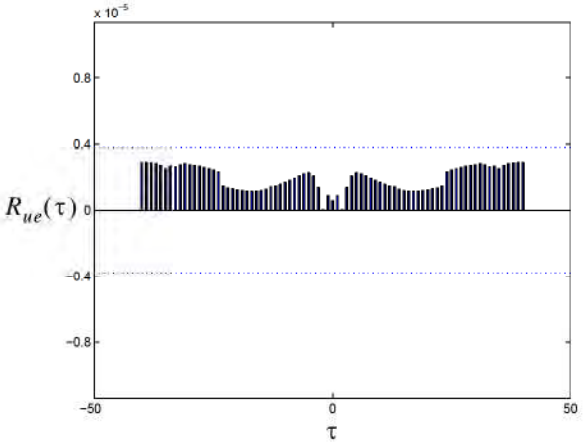


Figure 27.10 $R_{ue}(\tau)$ $(n_y = n_u = 2, S^1 = 10)$

Because the autocorrelation function of the prediction errors in Figure 27.9 indicated correlation in the errors, we increased the n_y and n_u values from 2 to 4 and retrained our neural network predictor. The resulting estimated autocorrelation function is shown in Figure 27.11. Here we can see that $R_e(\tau)$ falls within the confidence bounds, except at $\tau = 0$, which indicates that our model is performing correctly.

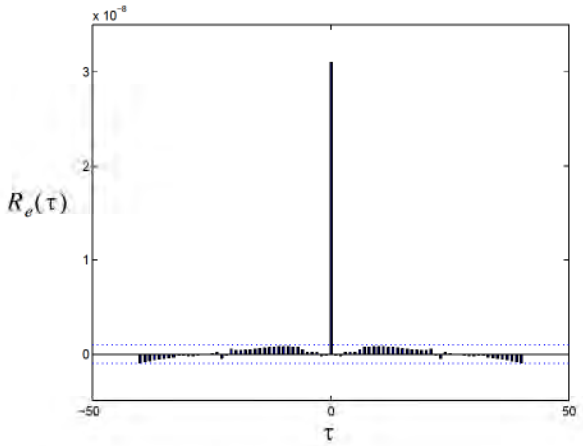


Figure 27.11 $R_e(\tau)$ $(n_y = n_u = 4, S^1 = 10)$

The estimated cross-correlation function, with the increased delay order, is shown in Figure 27.12. All of the points are well within the zero confidence

interval. There is no significant correlation between the errors and the input.

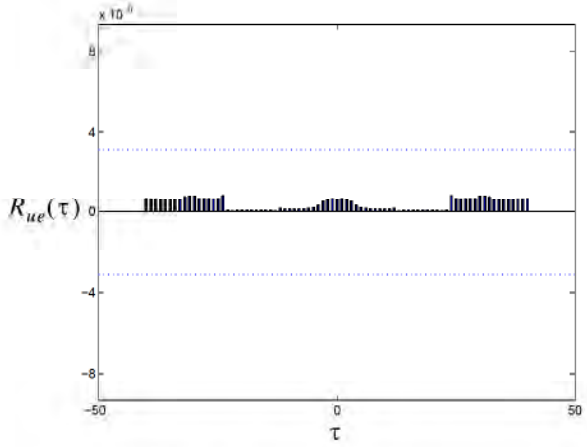


Figure 27.12 $R_{ue}(\tau)$ $(n_y = n_u = 4, S^1 = 10)$

With $n_y = n_u = 4$, we have white prediction errors, and there is no significant correlation between the prediction errors and the model input. It appears that we have an accurate prediction model.

The errors for the final prediction model are shown in Figure 27.13. We can see that the errors are very small. However, because of the series-parallel configuration, these are errors for only a one-step-ahead prediction. A more stringent test would be to rearrange the network into the original parallel form and then to perform an iterated prediction over many time steps. We will now demonstrate the parallel operation.

Figure 27.14 illustrates the iterated prediction. The solid line is the actual position of the magnet, and the dashed line is the position predicted by the NARX neural network. The network prediction is very accurate - even 600 time steps ahead.

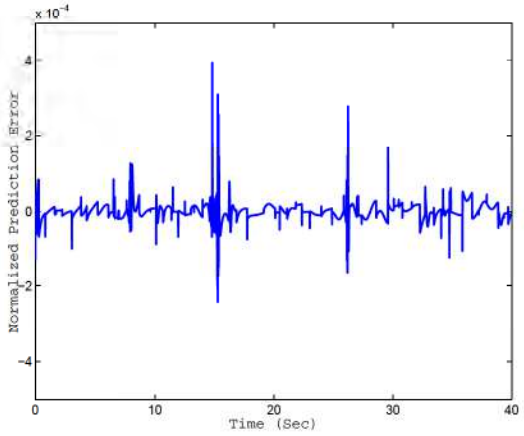


Figure 27.13 Prediction Errors vs. Time

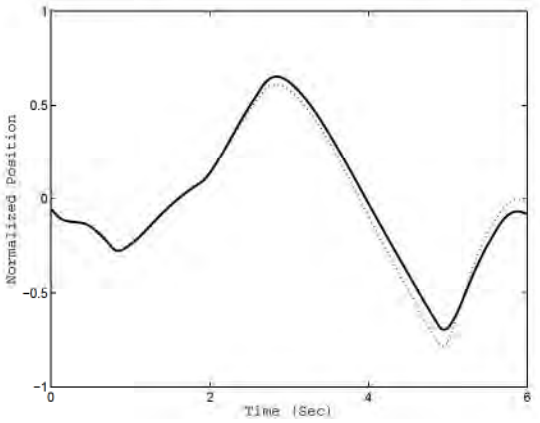


Figure 27.14 Iterated Prediction for the Maglev NARX Network

Data Sets

There are two data files associated with this case study:

- maglev_u.txt — contains the input sequence in the original data set
- maglev_y.txt — contains the output sequence in the original data set

They can be found with the demonstration software, which is described in Appendix C.

Epilogue

This chapter has demonstrated the use of multilayer neural networks for prediction, in which a future value of a time series is predicted from past values of that series and potentially other series. In this case study, the prediction network was used as a model of a magnetic levitation system. This modeling of dynamic systems is referred to as system identification.

A Nonlinear Autoregressive model with eXogenous inputs (NARX) network is well suited to this problem, and Bayesian regularization is an excellent training algorithm to use in this situation.

Further Reading

[BoJe94] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis: Forecasting and Control*, Fourth Edition, Wiley, 2008.

A classic textbook on time series analysis and the development of prediction models.

[HaDe02] M. Hagan, H. Demuth, O. De Jesus, "An Introduction to the Use of Neural Networks in Control Systems," *International Journal of Robust and Nonlinear Control*, vol. 12, no. 11, pp. 959-985, 2002.

This survey paper describes some practical aspects of using neural networks for control systems. Three neural network controllers are demonstrated: model predictive control, NARMA-L2 control, and model reference control.

[NaMu97] Narendra, K.S.; Mukhopadhyay, S., "Adaptive control using neural networks and approximate models," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 475 - 485, 1997.

This paper introduced the NARMA-L2 model and controller. Once the NARMA-L2 model is trained, it can be easily inverted to form a controller for the identified system.

[NaPa90] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.

Classic early paper on the use of neural networks for the identification and control of dynamical systems.

A Bibliography

[AgSa05] P.M. Agrawal, A.N.A. Samadh, L.M. Raff, M. Hagan, S. T. Bukkapatnam, and R. Komanduri, "Prediction of molecular-dynamics simulation results using feedforward neural networks: Reaction of a C2 dimer with an activated diamond (100) surface," *The Journal of Chemical Physics* 123, 224711, 2005. (Chapter 24)

[Albe72] A. Albert, *Regression and the Moore-Penrose Pseudoinverse*, New York: Academic Press, 1972. (Chapter 7)

[AmMu97] S. Amari, N. Murata, K.-R. Muller, M. Finke, and H. H. Yang, "Asymptotic Statistical Theory of Overtraining and Cross-Validation," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, 1997. (Chapter 13)

[Ande72] J. A. Anderson, "A simple neural network generating an interactive memory," *Mathematical Biosciences*, vol. 14, pp. 197–220, 1972. (Chapter 1, 15, 21)

[AnRo88] J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, Cambridge, MA: MIT Press, 1989. (Chapter 1, 10)

[AnSi77] J. A. Anderson, J. W. Silverstein, S. A. Ritz and R. S. Jones, "Distinctive features, categorical perception, and probability learning: Some applications of a neural model," *Psychological Review*, vol. 84, pp. 413–451, 1977. (Chapter 21)

[Barn92] E. Barnard, "Optimization for training neural nets," *IEEE Transactions on Neural Networks*, vol. 3, no. 2, pp. 232–240, 1992. (Chapter 12)

[BaSu83] A. R. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 13, pp. 834–846, 1983. (Chapter 4)

[Batt92] R. Battiti, "First and second order methods for learning: Between steepest descent and Newton's method," *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992. (Chapter 9, 12)

[Bish91] C. M. Bishop, "Improving the generalization properties of radial basis function neural networks," *Neural Computation*, vol. 3, no. 4, pp. 579-588, 1991. (Chapter 17)

[Bish95] C.M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press,1995. (Chapter 22)

[BlDe99] J. A. Blackard and D. J. Dean, "Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types from Cartographic Variables," Computers and Electronics in Agriculture, vol. 24, pp. 131-151, 1999. (Chapter 26)

[BoJe94] G.E.P. Box, G.M. Jenkins, and G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, 4th Edition, John Wiley & Sons, 2008. (Chapter 22, 27)

[BrLo88] D.S. Broomhead and D. Lowe, "Multivariable function interpolation and adaptive networks," *Complex Systems*, vol.2, pp. 321-355, 1988. (Chapter 17)

[Brog91] W. L. Brogan, *Modern Control Theory*, 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1991. (Chapter 4, 5, 6, 8, 9, 20)

[CaGr87a] G. A. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics, and Image Processing*, vol. 37, pp. 54–115, 1987. (Chapter 19)

[CaGr87b] G. A. Carpenter and S. Grossberg, "ART2: Self-organization of stable category recognition codes for analog input patterns," *Applied Optics*, vol. 26, no. 23, pp. 4919–4930, 1987. (Chapter 19)

[CaGr90] G. A. Carpenter and S. Grossberg, "ART3: Hierarchical search using chemical transmitters in self-organizing pattern recognition architectures," *Neural Networks*, vol. 3, no. 23, pp. 129–152, 1990. (Chapter 19)

[CaGrMa92] G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds and D. B. Rosen, "Fuzzy ARTMAP: A neural network architecture for incremental learning of analog multidimensional maps," *IEEE Transactions on Neural Networks*, vol. 3, pp. 698–713, 1992. (Chapter 19)

[CaGrRe91] G.A. Carpenter, S. Grossberg and J. Reynolds, "ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network," *Neural Networks*, vol. 4, pp. 565–588, 1991. (Chapter 19)

[Char92] C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEEE Proceedings*, vol. 139, no. 3, pp. 301–310, 1992. (Chapter 12)

[ChCo91] S. Chen, C.F.N. Cowan, and P.M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Transactions on Neural Networks*, vol.2, no.2, pp.302-309, 1991. (Chapter 17)

[ChCo92] S. Chen, P. M. Grant, and C. F. N. Cowan, "Orthogonal least squares algorithm for training multioutput radial basis function networks," *Proceedings of the Institute of Electrical Engineers*, vol. 139, Pt. F, no. 6, pp. 378–384, 1992. (Chapter 17)

[ChCh96] S. Chen, E. S. Chng, and K. Alkadhimi, "Regularised orthogonal least squares algorithm for constructing radial basis function networks," *International Journal of Control*, vol. 64, no. 5, pp. 829–837, 1996. (Chapter 17)

[ChCo99] S. Chen, C.F.N. Cowan, and P.M. Grant, "Combined Genetic Algorithm Optimization and Regularized Orthogonal Least Squares Learning for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, vol.10, no.5, pp.302-309, 1999. (Chapter 17)

[CoGr83] M. A. Cohen and S. Grossberg, "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 13, no. 5, pp. 815–826, 1983. (Chapter 20, 21)

[DARP88] *DARPA Neural Network Study*, Lexington, MA: MIT Lincoln Laboratory, 1988. (Chapter 1)

[DeHa07] O. De Jesús and M. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp., 2007. (Chapter 14)

[Dubi00] D. Dubin, *Rapid Interpretation of EKG's*, Sixth Edition, Tampa, FL: COVER, 2000. (Chapter 25)

[Fahl89] S. E. Fahlman, "Fast learning variations on back-propagation: An empirical study," in *Proceedings of the 1988 Connectionist Models Summer School*, D. Touretzky, G. Hinton and T. Sejnowski, eds., San Mateo, CA: Morgan Kaufmann, pp. 38–51, 1989. (Chapter 12)

[FoGi07] L. Fortuna, P. Giannone, S. Graziani, M. G. Xibilia, "Virtual Instruments Based on Stacked Neural Networks to Improve Product Quality Monitoring in a Refinery," *IEEE Transactions on Instrumentation and Measurement*, vol. 56, no. 1, pp. 95–101, 2007. (Chapter 23)

[FoHa97] D. Foresee and M. Hagan, "Gauss-Newton Approximation to Bayesian Learning," *Proceedings of the 1997 International Joint Conference on Neural Networks*, vol. 3, pp. 1930 - 1935, 1997. (Chapter 13)

[FrSk91] J. Freeman and D. Skapura, *Neural Networks: Algorithms, Applications, and Programming Techniques*, Reading, MA: Addison-Wesley, 1991. (Chapter 16)

[Gill81] P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981. (Chapter 8, 9)

[GoLa98] C. Goutte and J. Larsen, "Adaptive Regularization of Neural Networks Using Conjugate Gradient," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 2, pp. 1201-1204, 1998. (Chapter 13)

[GrMi89] S. Grossberg, E. Mingolla and D. Todorovic, "A neural network architecture for preattentive vision," *IEEE Transactions on Biomedical Engineering*, vol. 36, no. 1, pp. 65–84, Jan. 1989. (Chapter 18)

[Gros67] S. Grossberg, "Nonlinear difference-differential equations in prediction and learning theory," *Proceedings of the National Academy of Sciences*, vol. 58, pp. 1329–1334, 1967. (Chapter 21)

[Gros68] S. Grossberg, "Some physiological and biochemical consequences of psychological postulates," *Proceedings of the National Academy of Sciences*, vol. 60, pp. 758–765, 1968. (Chapter 15)

[Gros76] S. Grossberg, "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors," *Biological Cybernetics*, vol. 23, pp. 121–134, 1976. (Chapter 1, 18, 19)

[Gros80] S. Grossberg, "How does the brain build a cognitive code?" *Psychological Review*, vol. 88, pp. 375–407, 1980. (Chapter 1)

[Gros82] S. Grossberg, *Studies of Mind and Brain*, Boston: D. Reidel Publishing Co., 1982. (Chapter 15, 18, 19)

[Gros90] S. Grossberg, "Neural networks: From foundations to applications," Short-Course Notes, Boston University, Boston, MA, May 6–11, 1990. (Chapter 18)

[HaBo07] L. Hamm, B. W. Brorsen and M. T. Hagan, "Comparison of Stochastic Global Optimization Methods to Estimate Neu-

ral Network Weights," *Neural Processing Letters*, vol. 26, no. 3, December 2007. (Chapter 22)

[HaDe02] M. Hagan, H. Demuth, O. De Jesus, "An Introduction to the Use of Neural Networks in Control Systems," *International Journal of Robust and Nonlinear Control*, vol. 12, no. 11, pp. 959-985, 2002. (Chapter 27)

[HaMe94] M. T. Hagan and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994. (Chapter 12)

[HeBa99] S. Hettich and S. D. Bay, *The UCI KDD Archive* [http://kdd.ics.uci.edu], Irvine, CA: University of California, Department of Information and Computer Science, 1999. (Chapter 26)

[Hebb 49] D. O. Hebb, *The Organization of Behavior*, New York: Wiley, 1949. (Chapter 1, 7, 15)

[Hech90] R. Hecht-Nielsen, *Neurocomputing*, Reading, MA: Addison-Wesley, 1990. (Chapter 16)

[HeOh97] B. Hedén, H. Öhlin, R. Rittner, L. Edenbrandt, "Acute Myocardial Infarction Detected in the 12-Lead ECG by Artificial Neural Networks," *Circulation*, vol. 96, pp. 1798–1802, 1997. (Chapter 22)

[Himm72] D. M. Himmelblau, *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972. (Chapter 8, 9)

[Hopf82] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational properties," *Proceedings of the National Academy of Sciences*, vol. 79, pp. 2554–2558, 1982. (Chapter 1, 21)

[Hopf84] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences*, vol. 81, pp. 3088–3092, 1984. (Chapter 21)

[HoTa85] J. J. Hopfield and D. W. Tank, " 'Neural' computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–154, 1985. (Chapter 21)

[HoSt89] K. M. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. (Chapter 11)

[Hube88] D. H. Hubel, *Eye, Brain, and Vision*, New York: Scientific American Library, 1988. (Chapter 18)

[Jaco88] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, no. 4, pp 295–308, 1988. (Chapter 12)

[John01] G. L. Johnson, "Contributions to the comparative anatomy of the mammalian eye, chiefly based on ophthalmoscopic examination," *Philosophical Transactions of the Royal Society of London*, Series B., vol. 194, pp. 1–82, Plate 1, 1901. (Chapter 18)

[Joll02] I.T. Jolliffe, *Principal Component Analysis*, Springer Series in Statistics, 2nd ed., Springer, NY, 2002. (Chapter 22)

[Koho72] T. Kohonen, "Correlation matrix memories," *IEEE Transactions on Computers*, vol. 21, pp. 353–359, 1972. (Chapter 1, 15, 21)

[Koho87] T. Kohonen, *Self-Organization and Associative Memory*, 2nd Ed., Berlin: Springer-Verlag, 1987. (Chapter 15, 16)

[Koho93] T. Kohonen, "Things you haven't heard about the Self-Organizing Map," *Proceedings of the International Conference on Neural Networks* (ICNN), San Francisco, pp. 1147-1156, 1993. (Chapter 26)

[Koho95] T. Kohonen, Self-Organizing Map, 2nd ed., Springer-Verlag, Berlin, 1995. (Chapter 26)

[LaSa67] J. P. LaSalle, "An invariance principle in the theory of stability," in *Differential Equations and Dynamic Systems*, J. K. Hale and J. P. Lasalle, eds., New York: Academic Press, pp. 277–286, 1967. (Chapter 20)

[LeCu85] Y. Le Cun, "Une procedure d'apprentissage pour reseau a seuil assymetrique," *Cognitiva*, vol. 85, pp. 599–604, 1985. (Chapter 11)

[LeCu98] Y. LeCun, L. Bottou, G. B. Orr, K.-R. Mueller, "Efficient BackProp," *Lecture Notes in Computer Science*, vol. 1524, 1998. (Chapter 22)

[Leib90] D. Lieberman, *Learning, Behavior and Cognition*, Belmont, CA: Wadsworth, 1990. (Chapter 15)

[LiMi89] J. Li, A. N. Michel and W. Porod, "Analysis and synthesis of a class of neural networks: Linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, pp. 1405–1422, November 1989. (Chapter 21)

[Lowe89] D. Lowe, "Adaptive radial basis function nonlinearities, and the problem of generalization," *Proceedings of the First IEE International Conference on Artificial Neural Networks*, pp. 171 - 175, 1989. (Chapter 17)

[MacK92] D. J. C. MacKay, "Bayesian Interpolation," *Neural Computation*, vol. 4, pp. 415-447, 1992. (Chapter 13)

[MaNe99] J.R. Magnu and H. Neudecker, *Matrix Differential Calculus*, John Wiley & Sons, Ltd., Chichester, 1999. (Chapter 14)

[MaGa00] E. A. Maguire, D. G. Gadian, I. S. Johnsrude, C. D. Good, J. Ashburner, R. S. J. Frackowiak, and C. D. Frith, "Navigation-related structural change in the hippocampi of taxi drivers," Proceedings of the National Academy of Sciences, vol. 97, no. 8, pp. 4398-4403, 2000. (Chapter 1)

[McPi43] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943. (Chapter 1, 4, 21)

[Mill90] A.J. Miller, *Subset Selection in Regression*. Chapman and Hall, N.Y., 1990. (Chapter 17)

[Mill93] M.F. Miller, "A scaled conjugate gradient algorithm for fast supervised learning," Neural Networks, vol. 6, pp. 525-533, 1993. (Chapter 24)

[MoDa89] J. Moody and C.J. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, vol. 1, pp. 281–294, 1989. (Chapter 17)

[Moll93] M. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, pp. 525-533, 1993. (Chapter 22)

[MoMa01] G.B. Moody, R.G. Mark, and A.L. Goldberger, "PhysioNet: a Web-based resource for the study of physiologic signals," *IEEE Transactions on Engineering in Medicine and Biology*, vol. 20, no. 3, pp: 70-75, 2001. (Chapter 25)

[MiPa69] M. Minsky and S. Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969. (Chapter 1, 4)

[NaMu97] Narendra, K.S.; Mukhopadhyay, S., "Adaptive control using neural networks and approximate models," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 475 - 485, 1997. (Chapter 27)

[NaPa90] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990. (Chapter 27)

[NgWi90] D. Nguyen and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the IJCNN*, vol. 3, pp. 21–26, July 1990. (Chapter 12, 22)

[OrHa00] M. J. Orr, J. Hallam, A. Murray, and T. Leonard, "Assessing rbf networks using delve," IJNS, 2000. (Chapter 17)

[Park85] D. B. Parker, "Learning-logic: Casting the cortex of the human brain in silicon," Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985. (Chapter 11)

[PaSa93] J. Park and I.W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 5, pp. 305-316, 1993. (Chapter 17)

[PeCo93] M. P. Perrone and L. N. Cooper, "When networks disagree: Ensemble methods for hybrid neural networks," in *Neural Networks for Speech and Image Processing*, R. J. Mammone, Ed., Chapman-Hall, pp. 126-142, 1993. (Chapter 22)

[PhHa13] M. Phan and M. Hagan, "Error Surface of Recurrent Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 11, pp. 1709 - 1721, October, 2013. (Chapter 14)

[Powe87] M.J.D. Powell, "Radial basis functions for multivariable interpolation: a review," *Algorithms for Approximation*, pp. 143-167, Oxford, 1987. (Chapter 17)

[PuFe97] G.V. Puskorius and L.A. Feldkamp, "Extensions and enhancements of decoupled extended Kalman filter training," *Proceedings of the 1997 International Conference on Neural Networks*, vol. 3, pp. 1879-1883, 1997. (Chapter 22)

[RaMa05] L.M. Raff, M. Malshe, M. Hagan, D.I. Doughan, M.G. Rockley, and R. Komanduri, "*Ab initio* potential-energy surfaces for complex, multi-channel systems using modified novelty sampling and feedforward neural networks," *The Journal of Chemical Physics*, vol. 122, 2005. (Chapter 22, 24)

[RiIr90] A. K. Rigler, J. M. Irvine and T. P. Vogl, "Rescaling of variables in back propagation learning," *Neural Networks*, vol. 3, no. 5, pp 561–573, 1990. (Chapter 12)

[Rose58] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958. (Chapter 1, 4)

[Rose61] F. Rosenblatt, *Principles of Neurodynamics*, Washington DC: Spartan Press, 1961. (Chapter 4)

[RuHi86] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986. (Chapter 11)

[RuMc86] D. E. Rumelhart and J. L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, Cambridge, MA: MIT Press, 1986. (Chapter 1, 11, 16)

[Sarle95] W. S. Sarle, "Stopped training and other remedies for overfitting," In *Proceedings of the 27th Symposium on Interface*, 1995. (Chapter 13)

[Scal85] L. E. Scales, *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985. (Chapter 8, 9, 12)

[ScSm99] B. Schölkopf, A. Smola, K.-R. Muller, "Kernel Principal Component Analysis," in B. Schölkopf, C. J. C. Burges, A. J. Smola (Eds.), *Advances in Kernel Methods-Support Vector Learning*, MIT Press Cambridge, MA, USA, pp. 327-352, 1999. (Chapter 22)

[Shan90] D. F. Shanno, "Recent advances in numerical techniques for large-scale optimization," in *Neural Networks for Control*, Miller, Sutton and Werbos, eds., Cambridge, MA: MIT Press, 1990. (Chapter 12)

[SjLj94] J. Sjoberg and L. Ljung, "Overtraining, regularization and searching for minimum with application to neural networks," Linkoping University, Sweden, Tech. Rep. LiTH-ISY-R-1567, 1994. (Chapter 13)

[SlLi91] J.-J. E. Slotine and W. Li, *Applied Nonlinear Control*, Englewood Cliffs, NJ: Prentice-Hall, 1991. (Chapter 20)

[StDo84] W. D. Stanley, G. R. Dougherty and R. Dougherty, *Digital Signal Processing*, Reston VA: Reston Publishing Co., 1984. (Chapter 10)

[Stra76] G. Strang, *Linear Algebra and Its Applications*, New York: Academic Press, 1980. (Chapter 5, 6)

[TaHo86] D. W. Tank and J. J. Hopfield, "Simple 'neural' optimization networks: An A/D converter, signal decision circuit and a linear programming circuit," *IEEE Transactions on Circuits and Systems*, vol. 33, no. 5, pp. 533–541, 1986. (Chapter 21)

[TeSi00] J. B. Tenenbaum, V. de Silva, J. C. Langford, "A Global Geometric Framework for Nonlinear Dimensionality Reduction," *Science*, vol. 290, pp. 2319-2323, 2000. (Chapter 25)

[Tikh63] A. N. Tikhonov, "The solution of ill-posed problems and the regularization method," *Dokl. Acad. Nauk USSR*, vol. 151, no. 3, pp. 501-504, 1963. (Chapter 13)

[Toll90] T. Tollenaere, "SuperSAB: Fast adaptive back propagation with good scaling properties," *Neural Networks*, vol. 3, no. 5, pp. 561–573, 1990. (Chapter 12)

[vanT75] H. F. J. M. van Tuijl, "A new visual illusion: Neonlike color spreading and complementary color induction between subjective contours," *Acta Psychologica*, vol. 39, pp. 441–445, 1975. (Chapter 18)

[VoMa88] T. P. Vogl, J. K. Mangis, A. K. Zigler, W. T. Zink and D. L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, vol. 59, pp. 256–264, Sept. 1988. (Chapter 12)

[vond73] C. von der Malsburg, "Self-organization of orientation sensitive cells in the striate cortex," *Kybernetic*, vol. 14, pp. 85–100, 1973. (Chapter 18)

[WaVe94] C. Wang, S. S. Venkatesh, and J. S. Judd, "Optimal Stopping and Effective Machine Complexity in Learning," *Advances in Neural Information Processing Systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6, pp. 303-310, 1994. (Chapter 13)

[Werbo74] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. Thesis, Harvard University, Cambridge, MA, 1974. Also published as *The Roots of Backpropagation*, New York: John Wiley & Sons, 1994. (Chapter 11)

[Werb90] P. J. Werbos, "Backpropagation through time: What it is and how to do it," *Proceedings of the IEEE*, vol. 78, pp. 1550–1560, 1990. (Chapter 14)

[WeTe84] J. F. Werker and R. C. Tees, "Cross-language speech perception: Evidence for perceptual reorganization during the first year of life," Infant Behavior and Development, vol. 7, pp. 49-63, 1984. (Chapter 1)

[WhSo92] D. White and D. Sofge, eds., *Handbook of Intelligent Control*, New York:Van Nostrand Reinhold, 1992. (Chapter 4)

[WiHo60] B. Widrow, M. E. Hoff, "Adaptive switching circuits,"*1960 IRE WESCON Convention Record*, New York: IRE Part 4, pp. 96–104, 1960. (Chapter 1, 10)

[WiSt 85] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1985. (Chapter 10)

[WiWi 88] B. Widrow and R. Winter, "Neural nets for adaptive filtering and adaptive pattern recognition," *IEEE Computer Magazine*, March 1988, pp. 25–39. (Chapter 10)

[WiZi89] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, pp. 270–280, 1989. (Chapter 14)

B Notation

Basic Concepts

Scalars: small *italic* letters.....*a,b,c*

Vectors: small **bold** nonitalic letters.....**a,b,c**

Matrices: capital **BOLD** nonitalic letters.....**A,B,C**

Language

Vector means a column of numbers.

Row vector means a row of a matrix used as a vector (column).

General Vectors and Transformations (Chapters 5 and 6)

$x = A(y)$

Weight Matrices

Scalar Element

$w_{i,j}^{k}(t)$

i - row, j - column, k - layer, t - time or iteration

Matrix

$\mathbf{W}^{k}(t)$

Column Vector

$\mathbf{w}_{j}^{k}(t)$

Row Vector

${}_{i}\mathbf{w}^{k}(t)$

Bias Vector

Scalar Element

$b_{i}^{k}(t)$

Vector

$\mathbf{b}^{k}(t)$

Input Vector

Scalar Element

$p_i(t)$

As One of a Sequence of Input Vectors

$\mathbf{p}(t)$

As One of a Set of Input Vectors

$\mathbf{p}_q$

Net Input Vector

Scalar Element

$n_i^k(t)$ or $n_{i,q}^k$

Vector

$\mathbf{n}^k(t)$ or $\mathbf{n}_q^k$

Output Vector

Scalar Element

$a_i^k(t)$ or $a_{i,q}^k$

Vector

$\mathbf{a}^k(t)$ or $\mathbf{a}_q^k$

Transfer Function

Scalar Element

$a_i^k = f^k(n_i^k)$

Vector

$\mathbf{a}^k = \mathbf{f}^k(\mathbf{n}^k)$

Target Vector

Scalar Element

$t_i(t)$ or $t_{i,q}$

Vector

$\mathbf{t}(t)$ or $\mathbf{t}_q$

Set of Prototype Input/Target Vectors

$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$

Error Vector

Scalar Element

$e_i(t) = t_i(t) - a_i(t)$ or $e_{i,q} = t_{i,q} - a_{i,q}$

Vector

$\mathbf{e}(t)$ or $\mathbf{e}_q$

Sizes and Dimensions

Number of Layers, Number of Neurons per Layer

M, S^k

Number of Input Vectors (and Targets), Dimension of Input Vector

Q, R

Parameter Vector (includes all weights and biases)

Vector

$\mathbf{x}$

At Iteration k

$\mathbf{x}(k)$ or $\mathbf{x}_k$

Norm

$\|\mathbf{x}\|$

Performance Index

$F(\mathbf{x})$

Gradient and Hessian

$\nabla F(\mathbf{x}_k) = \mathbf{g}_k$ and $\nabla^2 F(\mathbf{x}_k) = \mathbf{A}_k$

Parameter Vector Change

$$\Delta \mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$$

Eigenvalue and Eigenvector

$$\lambda_i \text{ and } \mathbf{z}_i$$

Approximate Performance Index (single time step)

$$\hat{F}(\mathbf{x})$$

Transfer Function Derivative

Scalar

$$\dot{f}(n) = \frac{d}{dn} f(n)$$

Matrix

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}$$

Jacobian Matrix

$$\mathbf{J}(\mathbf{x})$$

Approximate Hessian Matrix

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

Sensitivity Vector

Scalar Element

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}$$

Vector

$$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m}$$

Marquardt Sensitivity Matrix

Scalar Element

$$\tilde{s}^m_{i,h} \equiv \frac{\partial v_h}{\partial n^m_{i,q}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}}$$

Partial Matrix (single input vector $\mathbf{p}_q$) and Full Matrix (all inputs)

$$\mathbf{S}^m_q \text{ and } \mathbf{S}^m = \left[\mathbf{S}^m_1\ \mathbf{S}^m_2\ \ldots\ \mathbf{S}^m_Q\right]$$

Dynamic Networks

Sensitivity

$$s^{u,m}_{k,i}(t) \equiv \frac{\partial^e a^u_k(t)}{\partial n^m_i(t)}$$

Weight Matrices

$\mathbf{IW}^{m,l}(d)$ - input weight between input l and layer m at delay d

$\mathbf{LW}^{m,l}(d)$ - layer weight between layer l and layer m at delay d

Index Sets

$DL_{m,l}$ - delays in the tapped delay line between Layer l and Layer m.

$DI_{m,l}$ - delays in the tapped delay line between Input l and Layer m.

I_m - indices of input vectors that connect to layer m.

L^f_m - indices of layers that directly connect *forward* to layer m.

L^b_m - indices of layers that are directly connected backwards to layer m (or to which layer m connects forward) and that contain no delays in the connection.

$$E^U_{LW}(x) = \{u \in U \ni \exists(\mathbf{LW}^{x,u}(d) \neq 0, d \neq 0)\}$$

$$E^X_S(u) = \{x \in X \ni \exists(\mathbf{S}^{u,x} \neq 0)\}$$

$$E_S(u) = \{x \ni \exists(\mathbf{S}^{u,x} \neq 0)\}$$

$$E^X_{LW}(u) = \{x \in X \ni \exists(\mathbf{LW}^{x,u}(d) \neq 0, d \neq 0)\}$$

$$E^U_S(x) = \{u \in U \ni \exists(\mathbf{S}^{u,x} \neq 0)\}$$

Definitions

Input Layer (X) - has an input weight, or contains any delays with any of its weight matrices

Output Layer (U) - its output will be compared to a target during training, or it is connected to an input layer through a matrix that has delays associated with it.

Parameters for Backpropagation and Variations

Learning Rate and Momentum

α and γ

Learning Rate Increase, Decrease and Percentage Change

η, ρ and ζ

Conjugate Gradient Direction Adjustment Parameter

β_k

Marquardt Parameters

μ and ϑ

Generalization

Regularization Parameters

α, β and $\rho = \frac{\alpha}{\beta}$

Effective Number of Parameters

γ

Selected Model

M

Sum Squared Error and Sum Squared Weights

E_D, E_W

Maximum Likelihood and Most Probable Weights

$\mathbf{x}^{ML}$, $\mathbf{x}^{MP}$

Feature Map Terms

Distance Between Neurons

d_{ij} - distance between neuron i and neuron j

Neighborhood

$$N_i(d) = \{j, d_{ij} \leq d\}$$

Grossberg and ART Networks

On-Center and Off-Surround Connection Matrices

$$^{+}\mathbf{W}^{1} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \text{ and } {}^{-}\mathbf{W}^{1} = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 \end{bmatrix}$$

Excitatory and Inhibitory Biases

$^{+}\mathbf{b}$ and $^{-}\mathbf{b}$

Time Constant

ε

Relative Intensity

$$\bar{p}_i = \frac{p_i}{\bar{P}} \text{ where } P = \sum_{i=1}^{S^1} p_j$$

Instar and Outstar Weight Matrices

$\mathbf{W}^{1:2}$ and $\mathbf{W}^{2:1}$

Orienting Subsystem Parameters

α, β and $\rho = \frac{\alpha}{\beta}$ (vigilance)

ART1 Learning Law Parameter

ζ

Lyapunov Stability

Lyapunov Function

$V(\mathbf{a})$

Zero Derivative Set, Largest Invariant Set and Closure

Z, L and L°

Bounded Lyapunov Function Set

$\Omega_{\eta} = \{\mathbf{a}: V(\mathbf{a}) < \eta\}$

Hopfield Network Parameters

Circuit Parameters

$T_{i,j}$, C, R_i, I_i, ρ

Amplifier Gain

γ

C Software

Introduction

We have used MATLAB, a numeric computation and visualization software package, in this text. However, MATLAB is not essential for using this book. The computer exercises can performed with any available programming language, and the *Neural Network Design Demonstrations*, while helpful, are not critical to understanding the material covered in this book.

MATLAB is widely available and, because of its matrix/vector notation and graphics, is a convenient environment in which to experiment with neural networks. We use MATLAB in two different ways. First, we have included a number of exercises for the reader to perform in MATLAB. Many of the important features of neural networks become apparent only for large scale problems, which are computationally intensive and not feasible for hand calculations. With MATLAB, neural network algorithms can be quickly implemented, and large scale problems can be tested conveniently. If MATLAB is not available, any other programming language can be used to perform the exercises.



The second way in which we use MATLAB is through the *Neural Network Design Demonstrations*, which can be downloaded from the website hagan.okstate.edu/nnd.html. These interactive demonstrations illustrate important concepts in each chapter. The icon to the left identifies references to these demonstrations in the text.

MATLAB, or the student edition of MATLAB, version 2010a or later, should be installed on your computer in a a folder named MATLAB. To create this directory or folder and complete the MATLAB installation process, follow the instructions given in the MATLAB documentation. Take care to follow the guidelines given for setting the path.

After the Neural Network Design Demonstration software has been loaded into the MATLAB directory on your computer (or if the MATLAB path has been set to include the directory containing thedemonstration software), the demonstrations can be invoked by typing **nnd** at the MATLAB prompt. All demonstrations are easily accessible from a master menu.

Overview of Demonstration Files

Running the Demonstrations

You can run the demonstrations directly by typing their names at the MATLAB prompt. Typing **help nndesign** brings up a list of all the demos you can choose from.

Alternatively, you can run the Neural Network Design splash window (**nnd**) and then click the Contents button. This will take you to a graphical Table of Contents. From there you can select chapters with buttons at the bottom of the window and individual demonstrations with popup menus.

Sound

Many of the demonstrations use sound. In many cases the sound adds to the understanding of a demonstration. In other cases it is there simply for fun. If you need to turn the sound off you can give MATLAB the following command and all demonstrations will run quietly:

```
nnsound off
```

To turn sound back on:

```
nnsound on
```

You may note that demonstrations that utilize sound often run faster when sound is off. In addition, on some machines which do not support sound errors can occur unless the sound is turned off.

List of Demonstrations

General

nnd - Splash screen.
nndtoc - Table of contents.
nnsound - Turn Neural Network Design sounds on and off.

Chapter 2, Neuron Model and Network Architectures

nnd2n1 - One-input neuron.
nnd2n2 - Two-input neuron.

Chapter 3, An Illustrative Example

nnd3pc - Perceptron classification.
nnd3hamc - Hamming classification.
nnd3hopc - Hopfield classification.

Chapter 4, Perceptron Learning Rule

nnd4db - Decision boundaries.
nnd4pr - Perceptron rule.

Chapter 5, Signal and Weight Vector Spaces

nnd5gs - Gram-Schmidt.
nnd5rb - Reciprocal basis.

Chapter 6, Linear Transformations for Neural Networks

nnd6lt - Linear transformations.
nnd6eg - Eigenvector game.

Chapter 7, Supervised Hebbian Learning

nnd7sh - Supervised Hebb.

Chapter 8, Performance Surfaces and Optimum Points

nnd8ts1 - Taylor series #1.
nnd8ts2 - Taylor series #2.
nnd8dd - Directional derivatives.
nnd8qf - Quadratic function.

Chapter 9, Performance Optimization

nnd9sdq - Steepest descent for quadratic function.
nnd9mc - Method comparison.
nnd9nm - Newton's method.
nnd9sd - Steepest descent.

Chapter 10, Widrow-Hoff Learning

nnd10nc - Adaptive noise cancellation.
nnd10eeg - Electroencephalogram noise cancellation.
nnd10lc - Linear pattern classification.

Chapter 11, Backpropagation

nnd11nf - Network function.
nnd11bc - Backpropagation calculation.
nnd11fa - Function approximation.
nnd11gn - Generalization.

Chapter 12, Variations on Backpropagation

nnd12sd1- Steepest descent backpropagation #1.
nnd12sd2 - Steepest descent backpropagation #2.
nnd12mo - Momentum backpropagation.
nnd12vl - Variable learning rate backpropagation.
nnd12ls - Conjugate gradient line search.
nnd12cg - Conjugate gradient backpropagation.
nnd12ms - Maquardt step.
nnd12m - Marquardt backpropagation.

Chapter 13, Generalization

nnd13es - Early stoppinng.
nnd13reg - Regularization.
nnd13breg - Bayesian regularization.
nnd13esr - Early stopping/regularization.

Chapter 14, Dynamic Networks

nnd14fir - Finite impulse response network.
nnd14iir - Infinite impulse response network.
nnd14dynd - Dynamic derivatives.
nnd14rnt - Recurrent network training.

Chapter 15, Associative Learning

nnd15uh - Unsupervised Hebb.
nnd15edr - Effect of decay rate.
nnd15hd - Hebb with decay.
nnd15gis - Graphical instar.
nnd15is - Instar.
nnd15os - Outstar.

Chapter 16, Competitive Networks

nnd16cc - Competitive classification.
nnd16cl - Competitive learning.
nnd16fm1 - 1-D feature map.
nnd16fm2 - 2-D feature map.
nnd16lv1 - LVQ1.
nnd16lv2 - LVQ2.

Chapter 17, Radial Basis Networks

nnd17nf - Network function.
nnd17pc - Pattern classification.
nnd17lls - Linear least squares.
nnd17ols - Orthogonal least squares.
nnd17no - Nonlinear optimization.

Chapter 18, Grossberg Network

nnd18li - Leaky integrator.
nnd18sn - Shunting network.
nnd18gl1 - Grossberg layer 1.
nnd18gl2 - Grossberg layer 2.
nnd18aw - Adaptive weights.

Chapter 19, Adaptive Resonance Theory

nnd19al1 - ART1 layer 1.
nnd19al2 - ART1 layer 2.
nnd19os - Orienting subsystem.
nnd19a1 - ART1 algorithm.

Chapter 20, Stability

nnd20ds - Dynamical system.

Chapter 21, Hopfield Network

nnd21hn - Hopfield network.

Index

A

Abbreviated notation 2-8
ADALINE network 10-2
  decision boundary 10-4
  mean square error 10-4
Adaptive filtering 10-13
Adaptive noise cancellation 10-15
Adaptive resonance theory (ART) 19-2
Amacrine cells 18-4
Amari, S. 18-2
AND gate 4-7
Anderson, J.A. 1-2, 1-3, 15-2, 18-2
Angle 5-7
Apple and Orange Example 3-2
  Hamming network solution 3-8
  Hopfield solution 3-12
  perceptron 3-3
  perceptron solution 3-5
  problem statement 3-2
Applications of neural networks 1-5
  aerospace 1-5
  automotive 1-5
  banking 1-5
  defense 1-6
  electronics 1-6
  entertainment 1-6
  financial 1-6
  insurance 1-6
  manufacturing 1-6
  medical 1-6
  oil and gas 1-6
  robotics 1-7
  securities 1-7
  speech 1-7
  telecommunications 1-7
  transportation 1-7
ART1
  fast learning 19-19
  Layer 1 19-4
  Layer 2 19-10
  learning law
    L1 - L2 19-17
    L2 - L1 19-20
  orienting subsystem 19-13
  resonance 19-17
  subset/superset dilemma 19-17
  summary 19-21
  vigilance 19-15
ART2 19-23
ART3 19-23
ARTMAP 19-23
Associative learning
  Hebb rule 7-4
  instar rule 15-11
  Kohonen rule 15-15
  outstar rule 15-17
  pseudoinverse rule 7-7
  unsupervised Hebb rule 15-5
Associative memory 7-3
  autoassociative memory 7-10
  Hopfield network 21-5
  linear associator 7-3
Associative networks 15-3
  instar 15-9
  outstar 15-16
Attractors 21-11
Autoassociative memory 7-10
Autocorrelation function 22-24

B

Backpropagation 11-7
  batching 12-7
  CGBP 12-15
  choice of network architecture 11-18
  conjugate gradient 12-14
  convergence 11-20

delta-bar-delta 12-13
drawbacks 12-3, 14-3
example 11-14
generalization 11-22
initial weights 12-6
Jacobian matrix 12-23
Levenberg-Marquardt 12-19, 12-21
Jacobian calculation 12-22
Marquardt sensitivity 12-24
LMBP 12-25
MOBP 12-11
performance index 11-8
performance surface 12-3
Quickprop 12-14
SDBP 12-2
sensitivity 11-10
summary 11-13
SuperSAB 12-14
VLBP 12-12
Backpropagation order 14-4
Backpropagation-through-time (BPTT) 14-2, 14-11, 14-22
Backward elimination 17-18
Basis set 5-5
Batch SOFM learning algorithm 26-6
Batch training 11-17
Batching 12-7
Bayes' rule 13-10
Bayes' Theorem 13-10
Bayesian analysis 13-12
effective number of parameters 13-16
evidence 13-13
Gauss-Newton approximation to Bayesian regularization 13-17
likelihood function 13-13
posterior density 13-13
prior density 13-13
Bayesian regularization 13-12, 23-6
effective number of parameters 23-6
Baysian analysis
most probable 13-14
Biological inspiration of neural networks 1-8
Bipolar cells 18-3
Brightness constancy 18-8

C

Carpenter, G. 19-2
Center 17-6
CGBP 12-15
Chain rule 11-9
Change of basis 6-6
similarity transformation 6-8
Chemical vapor deposition 24-2
Choice of network architecture 11-18
Circular hollow 8-16
Clustering 22-9
Coding the targets 22-7
Committee of networks 22-18, 25-10
Competitive learning 16-7
adaptive resonance theory 19-2
ART1 19-4
ART2 19-23
ART3 19-23
ARTMAP 19-23
Fuzzy ARTMAP 19-23
instar rule 16-7
Kohonen rule 16-7
learning rate 16-9
LVQ2 16-21
problems 16-9
Competitive networks 16-5
ART1 19-4
Grossberg 18-13
Hamming network 16-3
Lateral inhibition 16-5
learning vector quantization 16-16
self-organizing feature map 16-12
winner-take-all 16-5
Conditioned stimulus 15-3
Cones 18-3
Confusion matrix 22-21, 25-7
Conjugate directions 9-16
Conjugate gradient 9-15, 12-14
golden section search 12-17
interval location 12-16
interval reduction 12-16
Content-addressable memory 21-16
Contour plot 8-8
Contrast enhance 18-18

control
electromagnet 27-2
Correlation matrix 10-6
Cross-correlation function 22-25
Cross-entropy 22-17
Cross-validation 13-6
Curse of dimensionality 17-11

D
Decay rate 15-7
Decision boundary 4-5, 10-4, 11-4
Delay 2-13
Delta rule 7-13, 10-7
Delta-bar-delta 12-13
Descent direction 9-3
Diagonalization 6-13
Directional derivative 8-5
Distortion measure 22-23
Domain 6-2
Dynamic networks 14-2

E
Early stopping 13-7, 13-19, 25-7
Echo cancellation 10-21
EEG 10-15
Effective number of parameters 13-16, 13-23, 23-6
Eigenvalues 6-10
Eigenvectors 6-10
Electrocardiagram 25-2
electromagnet 27-2
Elliptical hollow 8-17
Emergent segmentation 18-6
Equilibrium point 20-4
Euclidean space 5-3
evidence 13-13
Excitatory 18-10
Extended Kalman filter algorithm 22-14
Extrapolation 13-3, 22-21, 22-27

F
Fahlman, A.E. 12-14
False negative 22-21
False positive 22-22
Featural filling-in 18-6
Feature extraction 22-6, 25-3
Finite impulse response 14-6
Fitting 22-8
Forest cover 26-2
Forward selection 17-18
Fovea 18-5
Fukushima, K. 18-2
Function approximation 11-4, 22-8
Fuzzy ARTMAP 19-23

G
Ganglion cells 18-4
Gauss-Newton algorithm 12-21
Jacobian matrix 12-20
Gauss-Newton approximation to Bayesian regularization 13-17
Generalization 11-22, 13-2
Golden Section search 12-17
Gradient 8-5
Gradient descent 9-2
Gram-Schmidt orthogonalization 5-8
Grossberg competitive network 18-13
choice of transfer function 18-20
Layer 1 18-13
Layer 2 18-17
learning law 18-22
relation to Kohonen law 18-24
Grossberg, S. 1-3, 15-2, 18-2, 19-2

H
Hamming network 3-8, 16-3
feedforward layer 3-8, 16-3
recurrent layer 3-9, 16-4
Hebb rule 7-4, 21-18
decay rate 15-7
performance analysis 7-5
supervised 7-4
unsupervised 7-4, 15-5
with decay 7-12
Hebb, D.O. 1-3, 7-2
Hebb's postulate 7-2
Hebbian learning 7-2
variations 7-12

Hessian 8-5
eigensystem 8-13
Hidden layer 2-11
High-gain Lyapunov function 21-13
Hinton, G.E. 11-2
Histogram of errors 22-21
History of neural networks 1-2
Hoff, M.E. 1-3, 10-2, 11-2
Hopfield model 21-3
Hopfield network 3-12, 6-2, 21-5
attractors 21-11
design 21-16
content-addressable memory 21-16
effect of gain 21-12
example 21-7
Hebb rule 21-18
high-gain Lyapunov function 21-13
Lasalle's invariance theorem 21-7
Lyapunov function 21-5
Lyapunov surface 21-22
spurious patterns 21-20
Hopfield, J.J. 1-4
Horizontal cells 18-4
Hubel, D.H. 16-2, 18-12

I

Illusions 18-4
Incremental training 11-17
Infinite impulse response 14-7
Inhibitory 18-10
Inner product 5-6
Input selection 22-12
Input weight 14-3
Instar 15-9
Instar rule 15-11, 16-7
Integrator 2-13
Interpolation 13-3
Interval location 12-16
Interval reduction 12-16
Invariant set 20-13

J

Jacobian matrix 12-20
Jacobs, R.A. 12-13

K

Kohonen rule 15-15, 16-7
graphical representation 16-7
Kohonen, T. 1-3, 15-2, 18-2

L

LaSalle's corollary 20-14
LaSalle's invariance theorem 20-13
invariant set 20-13
Set
L 20-13
Z 20-12
Lateral inhibition 16-5
Layer 2-9
competitive 16-5
problems 16-9
hidden 2-11
output layer 2-11
superscript 2-11
Layer weight 14-3
Layered Digital Dynamic Network (LDDN) 14-3
Le Cun, Y. 11-2
Leaky integrator 18-9
Learning rate 9-3, 10-8
competitive learning 16-9
stable 9-6, 10-10
Learning rules 4-2
ART1 19-21
backpropagation 11-7
competitive learning 16-7
delta rule 7-13
Grossberg competitive network 18-22
Hebb rule 7-4
Hebbian learning 7-2
learning vector quantization 16-16
LMS algorithm 10-7
local learning 15-5
perceptron 4-8, 4-13
proof of convergence 4-15
performance learning 8-2
pseudoinverse rule 7-7
reinforcement learning 4-3
supervised learning 4-3
unsupervised learning 4-3

Widrow-Hoff 7-13
Learning vector quantization (LVQ) 16-16
subclass 16-17
Levenberg-Marquardt algorithm 12-19, 12-21, 22-14, 24-7
Jacobian calculation 12-22
Jacobian matrix 12-20
Likelihood function 13-13
Linear associator 7-3
Linear independence 5-4
Linear initialization 26-6
Linear least squares 17-11
Linear separability 4-19
Linear transformation 6-2
change of basis 6-6
domain 6-2
matrix representation 6-3
change of basis 6-6
range 6-2
Linear vector spaces 5-2
LMBP 12-25
LMS algorithm 10-2, 10-7
adaptive filtering 10-13
adaptive noise cancellation 10-15
analysis of convergence 10-9
learning rate 10-8
stable learning rate 10-10
Local learning 15-5
Long term memory (LTM) 18-12, 18-22
LVQ2 16-21
Lyapunov function 20-12
Lyapunov stability theorem 20-6

M

Mach, E. 1-2
magnet 27-2
Marquardt algorithm 12-19
Marquardt sensitivity 12-24
Matrix representation 6-3
change of basis 6-6
diagonalization 6-13
McClelland, J.L. 1-4, 11-2
McCulloch, W.S. 1-3, 4-2
Mean square error 10-4, 11-8
Mean squared error 22-16
Memory
associative 7-3
autoassociative 7-10
Mexican-hat function 16-11
Minima 8-7
first-order conditions 8-10
global minimum 8-7
necessary conditions 8-9
second-order conditions 8-11
strong minimum 8-7
sufficient condition 8-11
weak minimum 8-7
Minkowski error 22-17
Minsky, M. 1-3, 4-2
Missing data 22-8
MOBP 12-11
Molecular dynamics 24-3
Momentum 12-9, 15-7
Monte Carlo simulation 25-9
Most probable 13-14
Multilayer perceptron 11-2
Myocardial infarction 25-2

N

NARX network 22-10
Negative definite matrix 8-11
Negative semidefinite 8-11
Neighborhood 16-12
Network architectures 2-9
layer 2-9
multiple layers 2-10
Neural Network Toolbox for MATLAB 1-5
Neuron model 2-2
multiple input neuron 2-7
single input neuron 2-2
transfer functions 2-3
Newton's method 9-10
Nguyen-Widrow weight initialization 22-13, 24-9
Nilsson, N. 16-2
Noise cancellation
adaptive 10-15
echo cancellation 10-21
Norm 5-7

Normalization 22-5, 25-6, 26-4
Novelty detection 22-28

O

Ockham's razor 13-2
On-center/off-surround 16-11, 18-14
Optic disk 18-5
Optimality
    first-order conditions 8-10
    necessary conditions 8-9
    second-order conditions 8-11
    sufficient condition 8-11
Optimization
    conjugate gradient 9-15, 12-14
    descent direction 9-3
    Gauss-Newton 12-21
    Levenberg-Marquardt 12-19, 12-21
    Newton's method 9-10
    quadratic termination 9-15
    steepest descent 9-2
        stable learning rates 9-6
Oriented receptive field 18-20
Orienting subsystem 19-13
Orthogonal least squares 17-18
Orthogonality 5-7
Orthonormal 5-9
Outliers 22-19
Outstar 15-16
Outstar rule 15-17
Overfitting 13-3, 22-27

P

Papert, S. 1-3, 4-2
Parker, D.B. 11-2
Pattern classification 11-3, 22-9, 25-2
Pavlov, I. 1-2
Perceptron 3-3
    architecture 4-3
    constructing learning rules 4-10
    decision boundary 4-5
    learning rule 4-8, 4-13
        proof of convergence 4-15
    multilayer 11-2
    multiple-neuron 4-8
    single-neuron 4-5
    test problem 4-9
    training multiple-neuron perceptrons 4-13
    two-input case 3-4
    unified learning rule 4-12
Performance Index 11-8
Performance index 8-2, 22-16
    cross-entropy 22-17
    mean squared error 22-16
    Minkowski error 22-17
    quadratic function 8-12
Performance learning 8-2
Pitts, W.H. 1-3, 4-2
Positive definite 20-5
Positive definite matrix 8-11
Positive semidefinite 8-11, 20-5
posterior density 13-13
Post-training analysis 22-18
Prediction 22-10, 22-24
Preprocessing 22-5
    coding the targets 22-7
    feature extraction 22-6
    normalization 22-5, 25-6, 26-4
    principal component analysis 22-6
Principal component analysis 22-6
Prior density 13-13
Probability estimation 24-2
Projection 5-8
Prototype patterns 21-16
Pseudoinverse rule 7-7

Q

Quadratic function 8-12
    circular hollow 8-16
    elliptical hollow 8-17
    Hessian
        eigensystem 8-13
    saddle point 8-18
    stationary valley 8-19
Quadratic termination 9-15
Quantization error 22-23
Quickprop 12-14

R

R value 22-20
Radial basis network 17-2
  backpropagation 17-25
  center 17-6
  pattern classification 17-6
Range 6-2
RBF 17-2
Real-time recurrent learning (RTRL) 14-2, 14-11, 14-12
Receiver operating characteristic (ROC) curve 22-22, 25-8
Reciprocal basis vectors 5-10
Recurrent 14-2
Recurrent network 2-13, 2-14, 20-2
regression 22-8
Regression/scatter plot 22-20
Regularization 13-8, 13-19, 13-21
Reinforcement learning 4-3
Resonance 19-17
Retina 18-3
Rods 18-3
Rosenblatt, F. 1-3, 4-2, 10-2, 11-2, 16-2
Rosenfeld, E. 1-2
Rumelhart, D.E. 1-4, 11-2

S

Saddle point 8-8, 8-18
Scaled conjugate gradient algorithm 22-14
SDBP 12-2
Segmentation 22-9
Self-organizing feature map (SOFM) 16-12, 22-16, 26-2
  distortion measure 22-23
  neighborhood 16-12
  quantization error 22-23
  topographic error 22-23
Sensitivity 11-10
  backpropagation 11-11
Sensitivity analysis 22-28
Set
  L 20-13
  Z 20-12
Shakespeare, W. 1-5
Short term memory (STM) 18-12, 18-17
Shunting model 18-10
Similarity transform 6-8
Simulation order 14-4
Smart sensor 23-2
Softmax 22-7, 24-6
Spanning a space 5-5
Spurious patterns 21-20
Stability
  asymptotically stable 20-3, 20-5
  concepts 20-3
  equilibrium point 20-4
  in the sense of Lyapunov 20-3, 20-4
  LaSalle's corollary 20-14
  LaSalle's invariance theorem 20-13
  Lyapunov function 20-12
  Lyapunov stability theorem 20-6
  pendulum example 20-6
Stability-plasticity dilemma 19-2
Stationary point 8-10
  minima 8-7
  saddle point 8-8
Stationary valley 8-19
Steepest descent 9-2
  learning rate 9-3
  minimizing along a line 9-8
  stable learning rates 9-6
Stimulus-response 15-2
  conditioned stimulus 15-3
  unconditioned stimulus 15-3
Stopping criteria 22-15
Subclass 16-17
Subset selection 17-18
Subset/superset dilemma 19-17
SuperSAB 12-14
Supervised learning 4-3
  Hebb rule 7-4
  performance learning 8-2
  target 4-3
  training set 4-3

T

Tapped delay line 10-13
Target 4-3

Taylor series expansion 8-2
vector case 8-4
Test set 13-6
Thomas Bayes 13-10
Tikhonov 13-8
Time constant 18-9
Tollenaere, T. 12-14
Topographic error 22-23
Training process 22-2
Training set 4-3
sequence 15-5
Transfer functions 2-3, 2-6
competitive 2-6
global vs local 17-9
hard limit 2-3, 2-6
hyperbolic tangent sigmoid 2-6
linear 2-4, 2-6
log-sigmoid 2-4, 2-6
positive linear 2-6
saturating linear 2-6
softmax 22-7, 24-6
symmetric saturating linear 2-6
symmetrical hard limit 2-6
table 2-6
Type I error 22-22

U

Unconditioned stimulus 15-3
Unsupervised learning 4-3
Hebb rule 7-4, 15-5

V

Validation set 13-7
Vector expansion 5-9
reciprocal basis vectors 5-10
Vector space 5-2
angle 5-7
basis set 5-5
orthonormal 5-9
projection 5-8
spanning 5-5
vector expansion 5-9
Vigilance 19-15
Vision 18-3
Vision normalization 18-8
Visual cortex 18-4
VLBP 12-12
von der Malsburg, C. 16-2, 18-12

W

Weight indices 2-7
Weight initialization 22-13, 24-9
Weight matrix 2-7
Werbos, P.J. 11-2
White noise 22-24
Widrow, B. 1-3, 10-2, 11-2
Widrow-Hoff algorithm 7-13, 10-7
adaptive filtering 10-13
Wiesel, T. 16-2, 18-12
Williams, R.J. 11-2
Winner-take-all 16-5

This book provides a clear and detailed coverage of fundamental neural network architectures and learning rules. In it, the authors emphasize a coherent presentation of the principal neural networks, methods for training them and their applications to practical problems.

Features

- Extensive coverage of training methods for both feedforward networks (including multilayer and radial basis networks) and recurrent networks. In addition to conjugate gradient and Levenberg-Marquardt variations of the backpropagation algorithm, the text also covers Bayesian regularization and early stopping, which ensure the generalization ability of trained networks.
- Associative and competitive networks, including feature maps and learning vector quantization, are explained with simple building blocks.
- A chapter of practical training tips for function approximation, pattern recognition, clustering and prediction, along with five chapters presenting detailed real-world case studies.
- Detailed examples and numerous solved problems. Slides and comprehensive demonstration software can be downloaded from hagan.okstate.edu/nnd.html.

About the Authors

Martin T. Hagan (Ph.D. Electrical Engineering, University of Kansas) has taught and conducted research in the areas of control systems and signal processing for the last 35 years. For the last 25 years his research has focused on the use of neural networks for control, filtering and prediction. He is a Professor in the School of Electrical and Computer Engineering at Oklahoma State University and a co-author of the Neural Network Toolbox for MATLAB.

Howard B. Demuth (Ph.D. Electrical Engineering, Stanford University) has twenty-three years of industrial experience, primarily at Los Alamos National Laboratory, where he helped design and build one of the world's first electronic computers, the "MANIAC." Demuth has fifteen years teaching experience as well. He is co-author of the Neural Network Toolbox for MATLAB and currently teaches a Neural Network course for the University of Colorado at Boulder.

Mark Hudson Beale (B.S. Computer Engineering, University of Idaho) is a software engineer with a focus on artificial intelligence algorithms and software development technology. Mark is co-author of the Neural Network Toolbox for MATLAB and provides related consulting through his company, MHB Inc., located in Hayden, Idaho.

Orlando De Jesús (Ph.D. Electrical Engineering, Oklahoma State University) has twenty-four years of industrial experience, with AETI C.A. in Caracas, Venezuela, Halliburton in Carrollton, Texas and is currently working as Engineering Consultant in Frisco, Texas. Orlando's dissertation was a basis for the dynamic network training algorithms in the Neural Network Toolbox for MATLAB.