of the chain gives the **depth** of the model. The name "deep learning" arose from this terminology. The final layer of a feedforward network is called the **output layer**. During neural network training, we drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$. The training data provides us with noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points. Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(\boldsymbol{x})$. The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers, they are called **hidden layers**.

Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representations is drawn from neuroscience. The choice of the functions $f^{(i)}(\boldsymbol{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. Modern neural network research, however, is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they can be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a

Microsoft Exhibit 1043 - Page 87
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

nonlinear transformation. Equivalently, we can apply the kernel trick described in section 5.7.2, to obtain a nonlinear learning algorithm based on implicitly applying the ϕ mapping. We can think of ϕ as providing a set of features describing $\boldsymbol{x}$, or as providing a new representation for $\boldsymbol{x}$.

The question is then how to choose the mapping ϕ.

1. One option is to use a very generic ϕ, such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel. If $\phi(\boldsymbol{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor. Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.

2. Another option is to manually engineer ϕ. Until the advent of deep learning, this was the dominant approach. It requires decades of human effort for each separate task, with practitioners specializing in different domains, such as speech recognition or computer vision, and with little transfer between domains.

3. The strategy of deep learning is to learn ϕ. In this approach, we have a model $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters $\boldsymbol{w}$ that map from $\phi(\boldsymbol{x})$ to the desired output. This is an example of a deep feedforward network, with ϕ defining a hidden layer. This approach is the only one of the three that gives up on the convexity of the training problem, but the benefits outweigh the harms. In this approach, we parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation. If we wish, this approach can capture the benefit of the first approach by being highly generic—we do so by using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$. Deep learning can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ that they expect will perform well. The advantage is that the human designer only needs to find the right general function family rather than finding precisely the right function.

This general principle of improving models by learning features extends beyond the feedforward networks described in this chapter. It is a recurring theme of deep learning that applies to all the kinds of models described throughout this book. Feedforward networks are the application of this principle to learning deterministic

Microsoft Exhibit 1043 - Page 88
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections. Other models, presented later, apply these principles to learning stochastic mappings, functions with feedback, and probability distributions over a single vector.

We begin this chapter with a simple example of a feedforward network. Next, we address each of the design decisions needed to deploy a feedforward network. First, training a feedforward network requires making many of the same design decisions as are necessary for a linear model: choosing the optimizer, the cost function, and the form of the output units. We review these basics of gradient-based learning, then proceed to confront some of the design decisions that are unique to feedforward networks. Feedforward networks have introduced the concept of a hidden layer, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values. We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer. Learning in deep neural networks requires computing the gradients of complicated functions. We present the **back-propagation** algorithm and its modern generalizations, which can be used to efficiently compute these gradients. Finally, we close with some historical perspective.

6.1 Example: Learning XOR

To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

The XOR function ("exclusive or") is an operation on two binary values, x_1 and x_2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(\boldsymbol{x})$ that we want to learn. Our model provides a function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$, and our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*.

In this simple example, we will not be concerned with statistical generalization. We want our network to perform correctly on the four points $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top$, and $[1, 1]^\top\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

We can treat this problem as a regression problem and use a mean squared error loss function. We have chosen this loss function to simplify the math for

Microsoft Exhibit 1043 - Page 89
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

this example as much as possible. In practical applications, MSE is usually not an appropriate cost function for modeling binary data. More appropriate approaches are described in section 6.2.2.2.

Evaluated on our whole training set, the MSE loss function is

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in\mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta}))^2 . \tag{6.1}$$

Now we must choose the form of our model, $f(\boldsymbol{x};\boldsymbol{\theta})$. Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of $\boldsymbol{w}$ and b. Our model is defined to be

$$f(\boldsymbol{x};\boldsymbol{w},b) = \boldsymbol{x}^\top \boldsymbol{w} + b. \tag{6.2}$$

We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\boldsymbol{w}$ and b using the normal equations.

After solving the normal equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}$. The linear model simply outputs 0.5 everywhere. Why does this happen? Figure 6.1 shows how a linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

Specifically, we will introduce a simple feedforward network with one hidden layer containing two hidden units. See figure 6.2 for an illustration of this model. This feedforward network has a vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c})$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$. The network now contains two functions chained together, $\boldsymbol{h} = f^{(1)}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h};\boldsymbol{w},b)$, with the complete model being $f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

What function should $f^{(1)}$ compute? Linear models have served us well so far, and it may be tempting to make $f^{(1)}$ linear as well. Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input. Ignoring the intercept terms for the moment, suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top \boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{W}^\top \boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}'$ where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$.

Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed nonlinear function called an activation function. We use that strategy here, by defining $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a

Microsoft Exhibit 1043 - Page 90
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

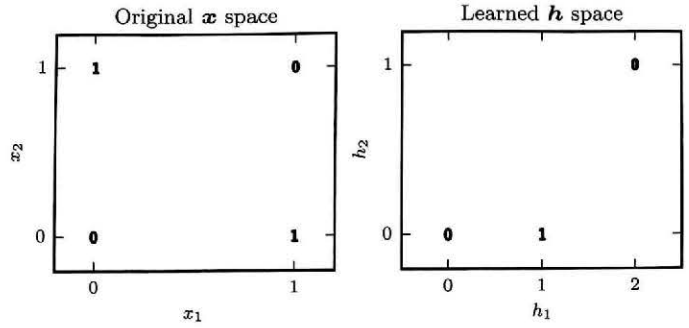


Figure 6.1: Solving the XOR problem by learning a representation. The bold numbers printed on the plot indicate the value that the learned function must output at each point. *(Left)* A linear model applied directly to the original input cannot implement the XOR function. When $x_1 = 0$, the model's output must increase as x_2 increases. When $x_1 = 1$, the model's output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2. The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem. *(Right)* In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the nonlinear features have mapped both $\boldsymbol{x} = [1, 0]^\top$ and $\boldsymbol{x} = [0, 1]^\top$ to a single point in feature space, $\boldsymbol{h} = [1, 0]^\top$. The linear model can now describe the function as increasing in h_1 and decreasing in h_2. In this example, the motivation for learning the feature space is only to make the model capacity greater so that it can fit the training set. In more realistic applications, learned representations can also help the model to generalize.

linear transformation and $\boldsymbol{c}$ the biases. Previously, to describe a linear regression model, we used a vector of weights and a scalar bias parameter to describe an affine transformation from an input vector to an output scalar. Now, we describe an affine transformation from a vector $\boldsymbol{x}$ to a vector $\boldsymbol{h}$, so an entire vector of bias parameters is needed. The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^\top \boldsymbol{W}_{:,i} + c_i)$. In modern neural networks, the default recommendation is to use the **rectified linear unit**, or ReLU (Jarrett et al., 2009; Nair and Hinton, 2010; Glorot et al., 2011a), defined by the activation function $g(z) = \max\{0, z\}$, depicted in figure 6.3.

We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b. \tag{6.3}$$

Microsoft Exhibit 1043 - Page 91
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

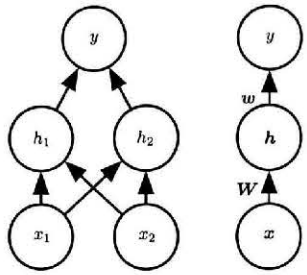


Figure 6.2: An example of a feedforward network, drawn in two different styles. Specifically, this is the feedforward network we use to solve the XOR example. It has a single hidden layer containing two units. *(Left)* In this style, we draw every unit as a node in the graph. This style is explicit and unambiguous, but for networks larger than this example, it can consume too much space. *(Right)* In this style, we draw a node in the graph for each entire vector representing a layer's activations. This style is much more compact. Sometimes we annotate the edges in this graph with the name of the parameters that describe the relationship between two layers. Here, we indicate that a matrix $\boldsymbol{W}$ describes the mapping from $\boldsymbol{x}$ to $\boldsymbol{h}$, and a vector $\boldsymbol{w}$ describes the mapping from $\boldsymbol{h}$ to y. We typically omit the intercept parameters associated with each layer when labeling this kind of drawing.

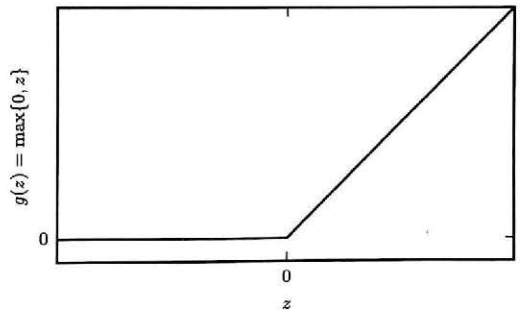


Figure 6.3: The rectified linear activation function. This activation function is the default activation function recommended for use with most feedforward neural networks. Applying this function to the output of a linear transformation yields a nonlinear transformation. The function remains very close to linear, however, in the sense that is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well. A common principle throughout computer science is that we can build complicated systems from minimal components. Much as a Turing machine's memory needs only to be able to store 0 or 1 states, we can build a universal function approximator from rectified linear functions.

Microsoft Exhibit 1043 - Page 92
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

We can then specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{6.4}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{6.5}$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \tag{6.6}$$

and $b = 0$.

We can now walk through how the model processes a batch of inputs. Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space, with one example per row:

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{6.7}$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{6.8}$$

Next, we add the bias vector $\boldsymbol{c}$, to obtain

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.9}$$

In this space, all the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function. To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.10}$$

Microsoft Exhibit 1043 - Page 93
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.

We finish with multiplying by the weight vector $\boldsymbol{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.11}$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here. Instead, a gradient-based optimization algorithm can find parameters that produce very little error. The solution we described to the XOR problem is at a global minimum of the loss function, so gradient descent could converge to this point. There are other equivalent solutions to the XOR problem that gradient descent could also find. The convergence point of gradient descent depends on the initial values of the parameters. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.

6.2 Gradient-Based Learning

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. In section 5.10, we described how to build a machine learning algorithm by specifying an optimization procedure, a cost function, and a model family.

The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become nonconvex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory—in practice it is robust but can encounter numerical problems). Stochastic gradient descent applied to nonconvex loss functions has no such convergence guarantee and is sensitive to the values of

Microsoft Exhibit 1043 - Page 94
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

Our first result derived using calculus of variations is that solving the optimization problem

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2 \tag{6.14}$$

yields

$$f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y}\sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}], \tag{6.15}$$

so long as this function lies within the class we optimize over. In other words, if we could train on infinitely many samples from the true data generating distribution, minimizing the mean squared error cost function would give a function that predicts the mean of $\boldsymbol{y}$ for each value of $\boldsymbol{x}$.

Different cost functions give different statistics. A second result derived using calculus of variations is that

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1 \tag{6.16}$$

yields a function that predicts the *median* value of $\boldsymbol{y}$ for each $\boldsymbol{x}$, as long as such a function may be described by the family of functions we optimize over. This cost function is commonly called **mean absolute error**.

Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization. Some output units that saturate produce very small gradients when combined with these cost functions. This is one reason that the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$.

6.2.2 Output Units

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

Any kind of neural network unit that may be used as an output can also be used as a hidden unit. Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well. We revisit these units with additional detail about their use as hidden units in section 6.3.

Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

Microsoft Exhibit 1043 - Page 95
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

6.2.2.1 Linear Units for Gaussian Output Distributions

One simple kind of output unit is based on an affine transformation with no nonlinearity. These are often just called linear units.

Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$.

Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}). \tag{6.17}$$

Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.

The maximum likelihood framework makes it straightforward to learn the covariance of the Gaussian too, or to make the covariance of the Gaussian be a function of the input. However, the covariance must be constrained to be a positive definite matrix for all inputs. It is difficult to satisfy such constraints with a linear output layer, so typically other output units are used to parametrize the covariance. Approaches to modeling the covariance are described shortly, in section 6.2.2.4.

Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

Many tasks require predicting the value of a binary variable y. Classification problems with two classes can be cast in this form.

The maximum likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$.

A Bernoulli distribution is defined by just a single number. The neural net needs to predict only $P(y = 1 \mid \boldsymbol{x})$. For this number to be a valid probability, it must lie in the interval $[0, 1]$.

Satisfying this constraint requires some careful design effort. Suppose we were to use a linear unit and threshold its value to obtain a valid probability:

$$P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\}. \tag{6.18}$$

This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent. Any time that $\boldsymbol{w}^\top \boldsymbol{h} + b$ strayed

Microsoft Exhibit 1043 - Page 96
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

outside the unit interval, the gradient of the output of the model with respect to its parameters would be $\mathbf{0}$. A gradient of $\mathbf{0}$ is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Instead, it is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer. This approach is based on using sigmoid output units combined with maximum likelihood.

A sigmoid output unit is defined by

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right), \tag{6.19}$$

where σ is the logistic sigmoid function described in section 3.10.

We can think of the sigmoid output unit as having two components. First, it uses a linear layer to compute $z = \boldsymbol{w}^\top \boldsymbol{h} + b$. Next, it uses the sigmoid activation function to convert z into a probability.

We omit the dependence on $\boldsymbol{x}$ for the moment to discuss how to define a probability distribution over y using the value z. The sigmoid can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, which does not sum to 1. We can then divide by an appropriate constant to obtain a valid probability distribution. If we begin with the assumption that the unnormalized log probabilities are linear in y and z, we can exponentiate to obtain the unnormalized probabilities. We then normalize to see that this yields a Bernoulli distribution controlled by a sigmoidal transformation of z:

$$\log \tilde{P}(y) = yz, \tag{6.20}$$

$$\tilde{P}(y) = \exp(yz), \tag{6.21}$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z),} \tag{6.22}$$

$$P(y) = \sigma\left((2y-1)z\right). \tag{6.23}$$

Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature. The z variable defining such a distribution over binary variables is called a **logit**.

This approach to predicting the probabilities in log space is natural to use with maximum likelihood learning. Because the cost function used with maximum likelihood is $-\log P(y \mid \boldsymbol{x})$, the log in the cost function undoes the exp of the sigmoid. Without this effect, the saturation of the sigmoid could prevent gradient-based

Microsoft Exhibit 1043 - Page 97
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

learning from making good progress. The loss function for maximum likelihood learning of a Bernoulli parametrized by a sigmoid is

$$J(\boldsymbol{\theta}) = -\log P(y \mid \boldsymbol{x}) \tag{6.24}$$

$$= -\log \sigma\left((2y-1)z\right) \tag{6.25}$$

$$= \zeta\left((1-2y)z\right). \tag{6.26}$$

This derivation makes use of some properties from section 3.10. By rewriting the loss in terms of the softplus function, we can see that it saturates only when $(1-2y)z$ is very negative. Saturation thus occurs only when the model already has the right answer—when $y = 1$ and z is very positive, or $y = 0$ and z is very negative. When z has the wrong sign, the argument to the softplus function, $(1-2y)z$, may be simplified to $|z|$. As $|z|$ becomes large while z has the wrong sign, the softplus function asymptotes toward simply returning its argument $|z|$. The derivative with respect to z asymptotes to $\text{sign}(z)$, so, in the limit of extremely incorrect z, the softplus function does not shrink the gradient at all. This property is useful because it means that gradient-based learning can act to quickly correct a mistaken z.

When we use other loss functions, such as mean squared error, the loss can saturate anytime $\sigma(z)$ saturates. The sigmoid activation function saturates to 0 when z becomes very negative and saturates to 1 when z becomes very positive. The gradient can shrink too small to be useful for learning when this happens, whether the model has the correct answer or the incorrect answer. For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units.

Analytically, the logarithm of the sigmoid is always defined and finite, because the sigmoid returns values restricted to the open interval $(0, 1)$, rather than using the entire closed interval of valid probabilities $[0, 1]$. In software implementations, to avoid numerical problems, it is best to write the negative log-likelihood as a function of z, rather than as a function of $\hat{y} = \sigma(z)$. If the sigmoid function underflows to zero, then taking the logarithm of $\hat{y}$ yields negative infinity.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function, which was used to represent a probability distribution over a binary variable.

Microsoft Exhibit 1043 - Page 98
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

In the case of binary variables, we wished to produce a single number

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}). \tag{6.27}$$

Because this number needed to lie between 0 and 1, and because we wanted the logarithm of the number to be well behaved for gradient-based optimization of the log-likelihood, we chose to instead predict a number $z = \log \tilde{P}(y = 1 \mid \boldsymbol{x})$. Exponentiating and normalizing gave us a Bernoulli distribution controlled by the sigmoid function.

To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. We require not only that each element of $\hat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the multinoulli distribution. First, a linear layer predicts unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

where $z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$. The softmax function can then exponentiate and normalize $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$. Formally, the softmax function is given by

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{6.29}$$

As with the logistic sigmoid, the use of the exp function works well when training the softmax to output a target value y using maximum log-likelihood. In this case, we wish to maximize $\log P(\text{y} = i; \boldsymbol{z}) = \log \text{softmax}(\boldsymbol{z})_i$. Defining the softmax in terms of exp is natural because the log in the log-likelihood can undo the exp of the softmax:

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j). \tag{6.30}$$

The first term of equation 6.30 shows that the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of equation 6.30 becomes very small. When maximizing the log-likelihood, the first

Microsoft Exhibit 1043 - Page 99
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

term encourages z_i to be pushed up, while the second term encourages all of $\boldsymbol{z}$ to be pushed down. To gain some intuition for the second term, $\log \sum_j \exp(z_j)$, observe that this term can be roughly approximated by $\max_j z_j$. This approximation is based on the idea that $\exp(z_k)$ is insignificant for any z_k that is noticeably less than $\max_j z_j$. The intuition we can gain from this approximation is that the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest input to the softmax, then the $-z_i$ term and the $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ terms will roughly cancel. This example will then contribute little to the overall training cost, which will be dominated by other examples that are not yet correctly classified.

So far we have discussed only a single example. Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set:

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i, \boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}. \tag{6.31}$$

Because maximum likelihood is a consistent estimator, this is guaranteed to happen as long as the model family is capable of representing the training distribution. In practice, limited model capacity and imperfect optimization will mean that the model is only able to approximate these fractions.

Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions (Bridle, 1990). To understand why these other loss functions can fail, we need to examine the softmax function itself.

Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. The softmax has multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

To see that the softmax function responds to the difference between its inputs, observe that the softmax output is invariant to adding the same scalar to all its inputs:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c). \tag{6.32}$$

Microsoft Exhibit 1043 - Page 100
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

Using this property, we can derive a numerically stable variant of the softmax:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i). \tag{6.33}$$

The reformulated version enables us to evaluate softmax with only small numerical errors, even when z contains extremely large or extremely negative numbers. Examining the numerically stable variant, we see that the softmax function is driven by the amount that its arguments deviate from $\max_i z_i$.

An output $\text{softmax}(\boldsymbol{z})_i$ saturates to 1 when the corresponding input is maximal ($z_i = \max_i z_i$) and z_i is much greater than all the other inputs. The output $\text{softmax}(\boldsymbol{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater. This is a generalization of the way that sigmoid units saturate and can cause similar difficulties for learning if the loss function is not designed to compensate for it.

The argument $\boldsymbol{z}$ to the softmax function can be produced in two different ways. The most common is simply to have an earlier layer of the neural network output every element of $\boldsymbol{z}$, as described above using the linear layer $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$. While straightforward, this approach actually overparametrizes the distribution. The constraint that the n outputs must sum to 1 means that only $n-1$ parameters are necessary; the probability of the n-th value may be obtained by subtracting the first $n-1$ probabilities from 1. We can thus impose a requirement that one element of $\boldsymbol{z}$ be fixed. For example, we can require that $z_n = 0$. Indeed, this is exactly what the sigmoid unit does. Defining $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$. Both the $n-1$ argument and the n argument approaches to the softmax can describe the same set of probability distributions but have different learning dynamics. In practice, there is rarely much difference between using the overparametrized version or the restricted version, and it is simpler to implement the overparametrized version.

From a neuroscientific point of view, it is interesting to think of the softmax as a way to create a form of competition between the units that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex. At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of **winner-take-all** (one of the outputs is nearly 1, and the others are nearly 0).

The name "softmax" can be somewhat confusing. The function is more closely related to the arg max function than the max function. The term "soft" derives from the fact that the softmax function is continuous and differentiable. The

Microsoft Exhibit 1043 - Page 101
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

arg max function, with its result represented as a one-hot vector, is not continuous or differentiable. The softmax function thus provides a "softened" version of the arg max. The corresponding soft version of the maximum function is $\text{softmax}(\boldsymbol{z})^\top \boldsymbol{z}$. It would perhaps be better to call the softmax function "softargmax," but the current name is an entrenched convention.

6.2.2.4 Other Output Types

The linear, sigmoid, and softmax output units described above are the most common. Neural networks can generalize to almost any kind of output layer that we wish. The principle of maximum likelihood provides a guide for how to design a good cost function for nearly any kind of output layer.

In general, if we define a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, the principle of maximum likelihood suggests we use $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as our cost function.

In general, we can think of the neural network as representing a function $f(\boldsymbol{x}; \boldsymbol{\theta})$. The outputs of this function are not direct predictions of the value $\boldsymbol{y}$. Instead, $f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ provides the parameters for a distribution over y. Our loss function can then be interpreted as $-\log p(\mathbf{y}; \boldsymbol{\omega}(\boldsymbol{x}))$.

For example, we may wish to learn the variance of a conditional Gaussian for $\mathbf{y}$, given $\mathbf{x}$. In the simple case, where the variance σ^2 is a constant, there is a closed form expression because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations $\mathbf{y}$ and their expected value. A computationally more expensive approach that does not require writing special-case code is to simply include the variance as one of the properties of the distribution $p(\mathbf{y} \mid \boldsymbol{x})$ that is controlled by $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The negative log-likelihood $-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ will then provide a cost function with the appropriate terms necessary to make our optimization procedure incrementally learn the variance. In the simple case where the standard deviation does not depend on the input, we can make a new parameter in the network that is copied directly into $\boldsymbol{\omega}$. This new parameter might be σ itself or could be a parameter v representing σ^2 or it could be a parameter β representing $\frac{1}{\sigma^2}$, depending on how we choose to parametrize the distribution. We may wish our model to predict a different amount of variance in $\mathbf{y}$ for different values of $\mathbf{x}$. This is called a **heteroscedastic** model. In the heteroscedastic case, we simply make the specification of the variance be one of the values output by $f(\mathbf{x}; \boldsymbol{\theta})$. A typical way to do this is to formulate the Gaussian distribution using precision, rather than variance, as described in equation 3.22. In the multivariate case, it is most common to use a diagonal precision matrix

$$\text{diag}(\boldsymbol{\beta}). \tag{6.34}$$

Microsoft Exhibit 1043 - Page 102
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

that is, exponential in depth l. In the case of maxout networks with k filters per unit, the number of linear regions is

$$O\left(k^{(l-1)+d}\right). \tag{6.43}$$

Of course, there is no guarantee that the kinds of functions we want to learn in applications of machine learning (and in particular for AI) share such a property.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks (Bengio et al., 2007; Erhan et al., 2009; Bengio, 2009; Mesnil et al., 2011; Ciresan et al., 2012; Krizhevsky et al., 2012; Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013; Kahou et al., 2013; Goodfellow et al., 2014d; Szegedy et al., 2014a). See figure 6.6 and figure 6.7 for examples of some of these empirical results. These results suggest that using deep architectures does indeed express a useful prior over the space of functions the model learns.

6.4.2 Other Architectural Considerations

So far we have described neural networks as being simple chains of layers, with the main considerations being the depth of the network and the width of each layer. In practice, neural networks show considerably more diversity.

Many neural network architectures have been developed for specific tasks. Specialized architectures for computer vision called convolutional networks are described in chapter 9. Feedforward networks may also be generalized to the recurrent neural networks for sequence processing, described in chapter 10, which have their own architectural considerations.

Microsoft Exhibit 1043 - Page 103
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

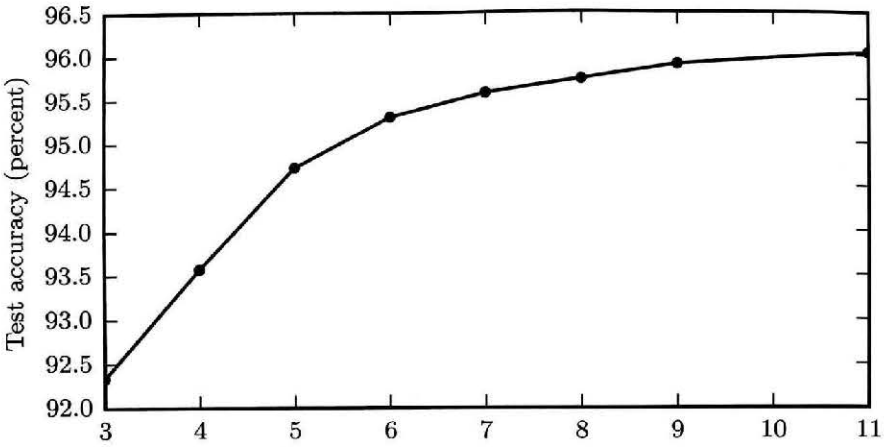


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow et al. (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

In general, the layers need not be connected in a chain, even though this is the most common practice. Many architectures build a main chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i+2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.

Another key consideration of architecture design is exactly how to connect a pair of layers to each other. In the default neural network layer described by a linear transformation via a matrix $\boldsymbol{W}$, every input unit is connected to every output unit. Many specialized networks in the chapters ahead have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer. These strategies for decreasing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network but are often highly problem dependent. For example, convolutional networks, described in chapter 9, use specialized patterns of sparse connections that are very effective for computer vision problems. In this chapter, it is difficult to give more specific advice concerning the architecture of a generic neural network. In subsequent chapters we develop the particular architectural strategies that have been found to work well for different application domains.

Microsoft Exhibit 1043 - Page 104
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

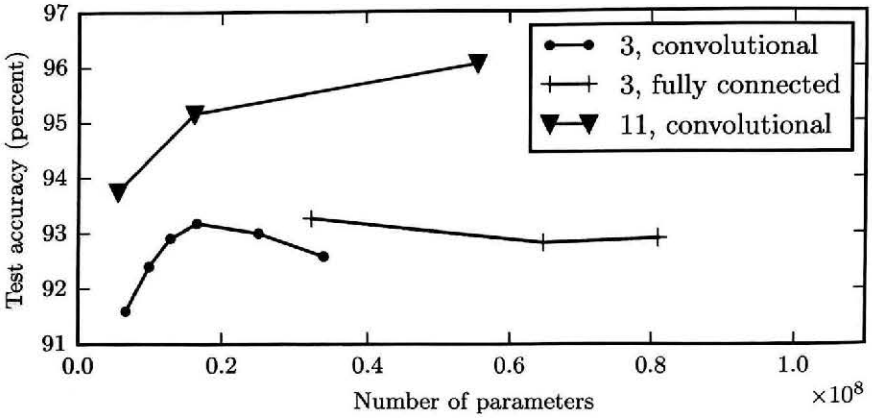


Figure 6.7: Effect of number of parameters. Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow et al. (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance, as illustrated in this figure. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

6.5 Back-Propagation and Other Differentiation Algorithms

When we use a feedforward neural network to accept an input $\boldsymbol{x}$ and produce an output $\hat{\boldsymbol{y}}$, information flows forward through the network. The input $\boldsymbol{x}$ provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$. This is called **forward propagation**. During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The **back-propagation** algorithm (Rumelhart et al., 1986a), often simply called **backprop**, allows the information from the cost to then flow backward through the network in order to compute the gradient.

Microsoft Exhibit 1043 - Page 105
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined). Specifically, we will describe how to compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ is an additional set of variables that are inputs to the function but whose derivatives are not required. In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Many machine learning tasks involve computing other derivatives, either as part of the learning process, or to analyze the learned model. The back-propagation algorithm can be applied to these tasks as well and is not restricted to computing the gradient of the cost function with respect to the parameters. The idea of computing derivatives by propagating information through a network is very general and can be used to compute values such as the Jacobian of a function f with multiple outputs. We restrict our description here to the most commonly used case, where f has a single output.

6.5.1 Computational Graphs

So far we have discussed neural networks with a relatively informal graph language. To describe the back-propagation algorithm more precisely, it is helpful to have a more precise **computational graph** language.

Many ways of formalizing computation as graphs are possible.

Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.

To formalize our graphs, we also need to introduce the idea of an **operation**. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together.

Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple

Microsoft Exhibit 1043 - Page 106
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

entries, such as a vector. Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case in our description because it introduces many extra details that are not important to conceptual understanding.

If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y. We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Examples of computational graphs are shown in figure 6.8.

6.5.2 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Usually we apply the back-propagation algorithm to tensors of arbitrary dimensionality, not merely to vectors. Conceptually, this is exactly the same as

Microsoft Exhibit 1043 - Page 107
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

9

Convolutional Networks

Convolutional networks (LeCun, 1989), also known as **convolutional neural networks**, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called **convolution**. Convolution is a specialized kind of linear operation. *Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*

In this chapter, we first describe what convolution is. Next, we explain the motivation behind using convolution in a neural network. We then describe an operation called **pooling**, which almost all convolutional networks employ. Usually, the operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields, such as engineering or pure mathematics. We describe several variants on the convolution function that are widely used in practice for neural networks. We also show how convolution may be applied to many kinds of data, with different numbers of dimensions. We then discuss means of making convolution more efficient. Convolutional networks stand out as an example of neuroscientific principles influencing deep learning. We discuss these neuroscientific principles, then conclude with comments about the role convolutional networks have played in the history of deep learning. One topic this chapter does not address is how to choose the architecture of your convolutional network. The goal of this chapter is to describe the kinds of tools that convolutional networks provide, while chapter 11 describes general guidelines

Microsoft Exhibit 1043 - Page 108
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

for choosing which tools to use in which circumstances. Research into convolutional network architectures proceeds so rapidly that a new best architecture for a given benchmark is announced every few weeks to months, rendering it impractical to describe in print the best architecture. Nonetheless, the best architectures have consistently been composed of the building blocks described here.

9.1 The Convolution Operation

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time t. Both x and t are real valued, that is, we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da. \tag{9.1}$$

This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t). \tag{9.2}$$

In our example, w needs to be a valid probability density function, or the output will not be a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example, though. In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input**, and the second

Microsoft Exhibit 1043 - Page 109
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

argument (in this example, the function w) as the **kernel**. The output is sometimes referred to as the **feature map**.

In our example, the idea of a laser sensor that can provide measurements at every instant is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{9.3}$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but in the finite set of points for which we store the values. This means that in practice, we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{9.4}$$

Convolution is commutative, meaning we can equivalently write

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{9.5}$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n.

The commutative property of convolution arises because we have **flipped** the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property

Microsoft Exhibit 1043 - Page 110
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the **cross-correlation**, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n). \tag{9.6}$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we follow this convention of calling both operations convolution and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel that is flipped relative to the kernel learned by an algorithm without the flipping. It is also rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not.

See figure 9.1 for an example of convolution (without kernel flipping) applied to a 2-D tensor.

Discrete convolution can be viewed as multiplication by a matrix, but the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This is known as a **Toeplitz matrix**. In two dimensions, a **doubly block circulant matrix** corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution, without requiring any further changes to the neural network. Typical convolutional neural networks do make use of further specializations in order to deal with large inputs efficiently, but these are not strictly necessary from a theoretical perspective.

9.2 Motivation

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant**

Microsoft Exhibit 1043 - Page 111
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

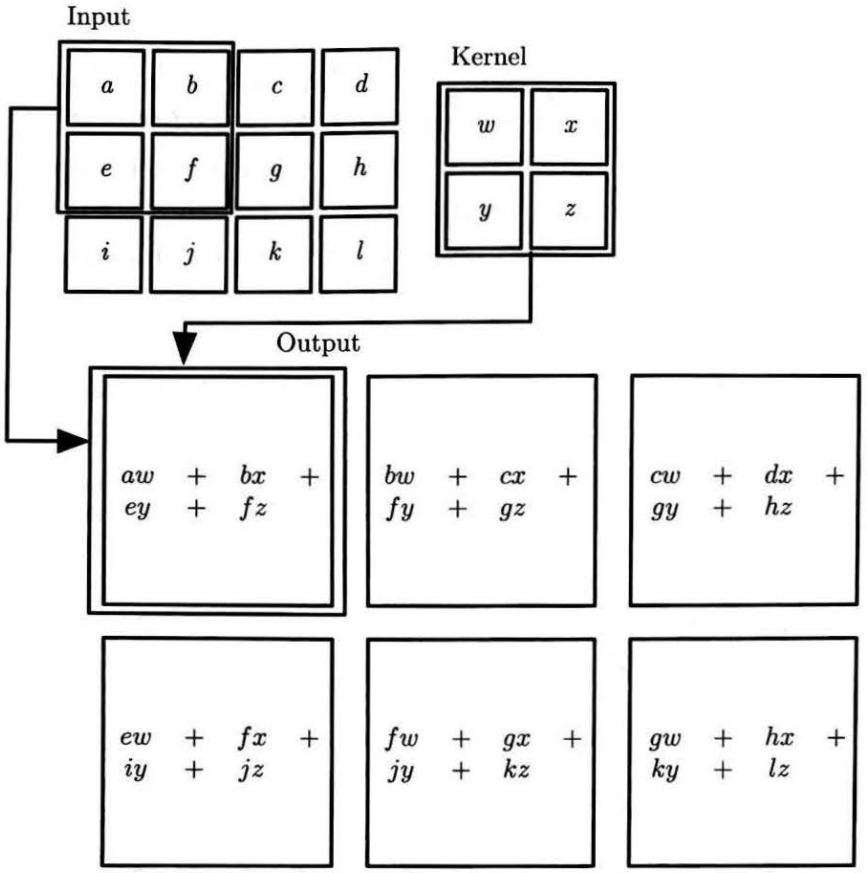


Figure 9.1: An example of 2-D convolution without kernel flipping. We restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

representations. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have **sparse interactions** (also referred to as **sparse connectivity** or **sparse weights**). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output

Microsoft Exhibit 1043 - Page 112
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m. For graphical demonstrations of sparse connectivity, see figure 9.2 and figure 9.3. In a deep convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input, as shown in figure 9.4. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix

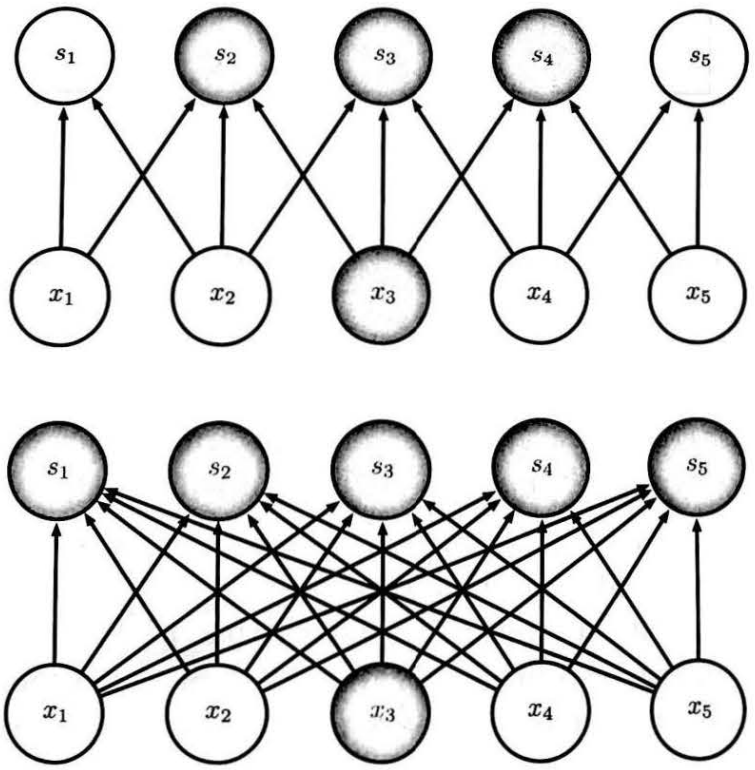


Figure 9.2: Sparse connectivity, viewed from below. We highlight one input unit, x_3, and highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by x_3.

Microsoft Exhibit 1043 - Page 113
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

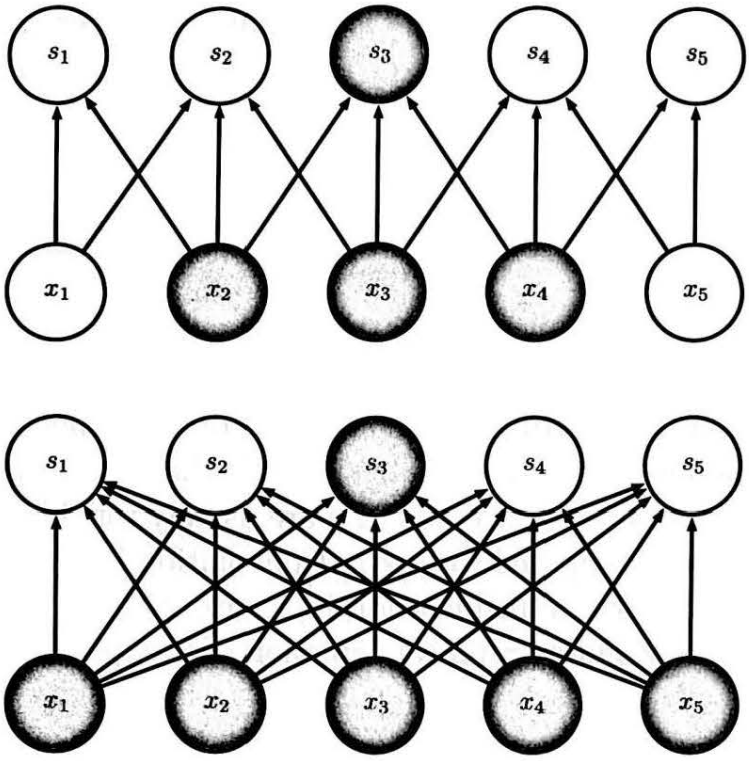


Figure 9.3: Sparse connectivity, viewed from above. We highlight one output unit, s_3, and highlight the input units in $\boldsymbol{x}$ that affect this unit. These units are known as the **receptive field** of s_3. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three inputs affect s_3. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all the inputs affect s_3.

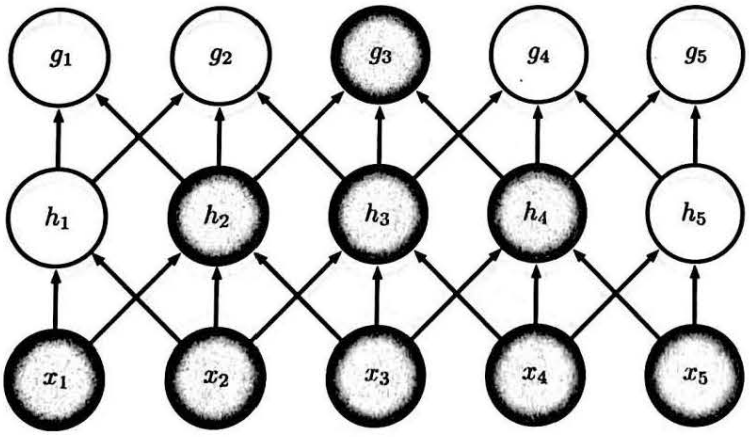


Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (figure 9.12) or pooling (section 9.3). This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

Microsoft Exhibit 1043 - Page 114
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has **tied weights**, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still $O(k \times n)$—but it does further reduce the storage requirements of the model to k parameters. Recall that k is usually several orders of magnitude smaller than m. Since m and n are usually roughly the same size, k is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see figure 9.5.

As an example of both of these first two principles in action, figure 9.6 shows how sparse connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image.

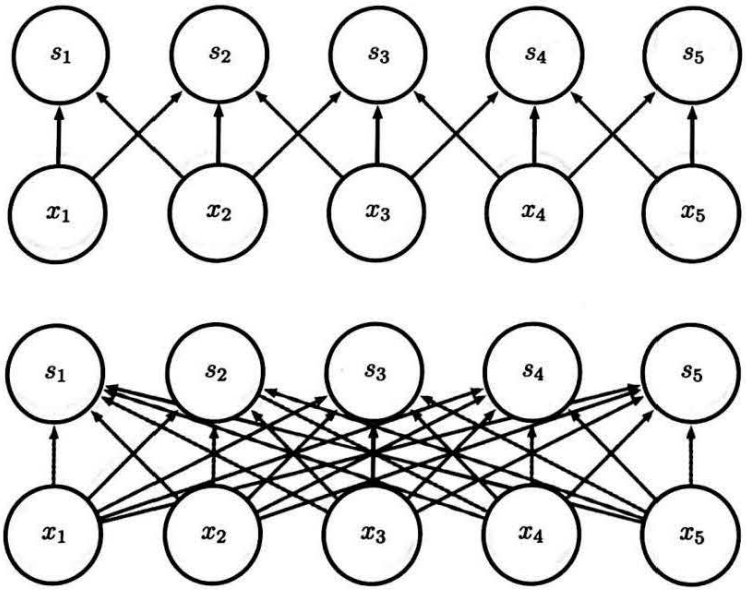


Figure 9.5: Parameter sharing. Black arrows indicate the connections that use a particular parameter in two different models. *(Top)*The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Because of parameter sharing, this single parameter is used at all input locations. *(Bottom)*The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing, so the parameter is used only once.

Microsoft Exhibit 1043 - Page 115
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

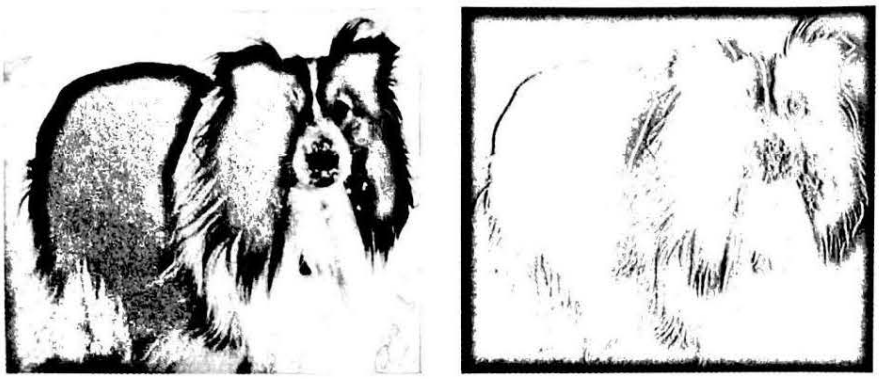


Figure 9.6: Efficiency of edge detection. The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide, while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing two elements, and requires $319 \times 280 \times 3 = 267,960$ floating-point operations (two multiplications and one addition per output pixel) to compute using convolution. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over eight billion, entries in the matrix, making convolution four billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over sixteen billion floating point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating-point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small local region across the entire input. Photo credit: Paula Goodfellow.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance** to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g. For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x, y) = I(x - 1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I', then applied the transformation g to the output. When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input.

Microsoft Exhibit 1043 - Page 116
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

If we move an event later in time in the input, the exact same representation of it will appear in the output, just later. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual's face, we probably want to extract different features at different locations—the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

Finally, some kinds of data cannot be processed by neural networks defined by matrix multiplication with a fixed-shape matrix. Convolution enables processing of some of these kinds of data. We discuss this further in section 9.7.

9.3 Pooling

A typical layer of a convolutional network consists of three stages (see figure 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the **detector stage**. In the third stage, we use a **pooling function** to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the **max pooling** (Zhou and Chellappa, 1988) operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation approximately **invariant** to small translations of the input. Invariance to translation means that if we

Microsoft Exhibit 1043 - Page 117
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

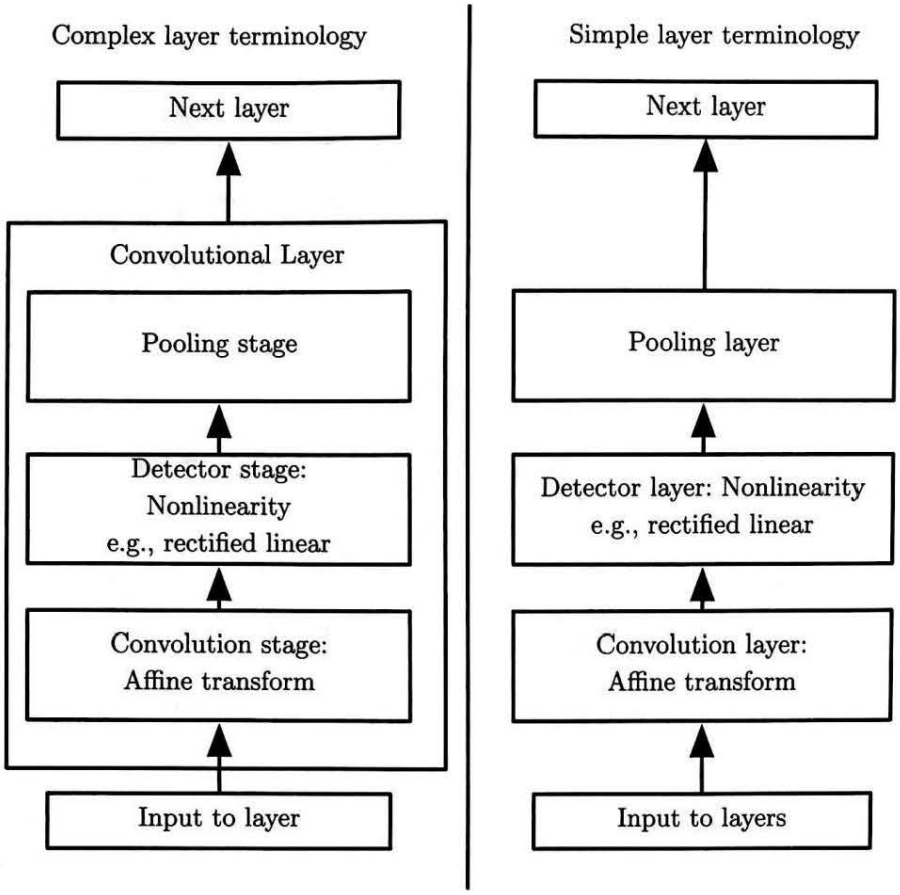


Figure 9.7: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. *(Left)*In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many "stages." In this terminology, there is a one-to-one mapping between kernel tensors and network layers. In this book we generally use this terminology. *(Right)*In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every "layer" has parameters.

translate the input by a small amount, the values of most of the pooled outputs do not change. See figure 9.8 for an example of how this works. *Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.* For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature. For example, if we want to find a corner defined by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet.

Microsoft Exhibit 1043 - Page 118
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

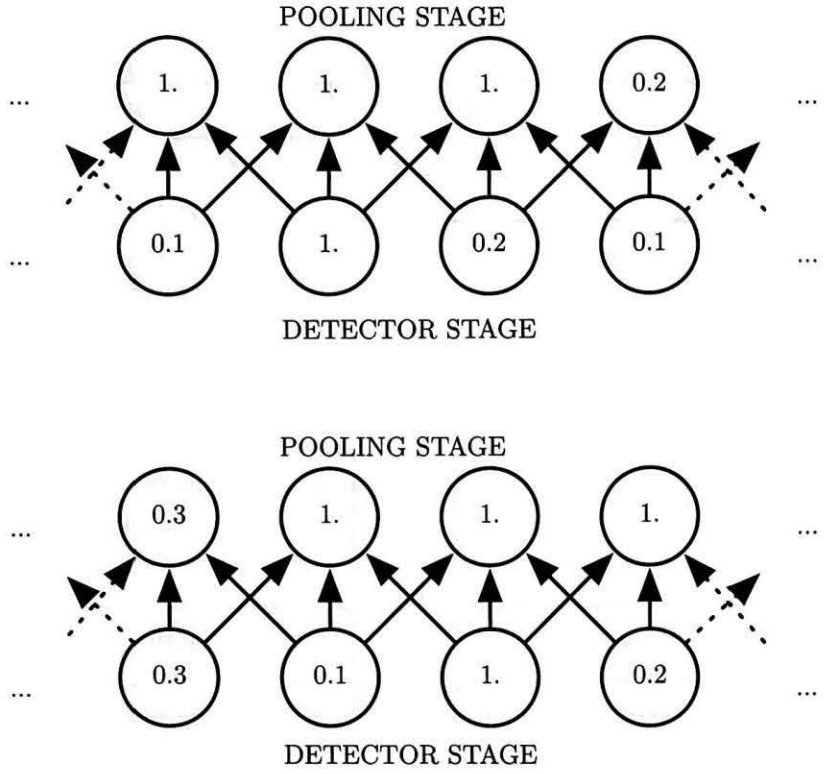


Figure 9.8: Max pooling introduces invariance. *(Top)*A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. *(Bottom)*A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are sensitive only to the maximum value in the neighborhood, not its exact location.

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (see figure 9.9).

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k pixels apart rather than 1 pixel apart. See figure 9.10 for an example. This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as

Microsoft Exhibit 1043 - Page 119
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

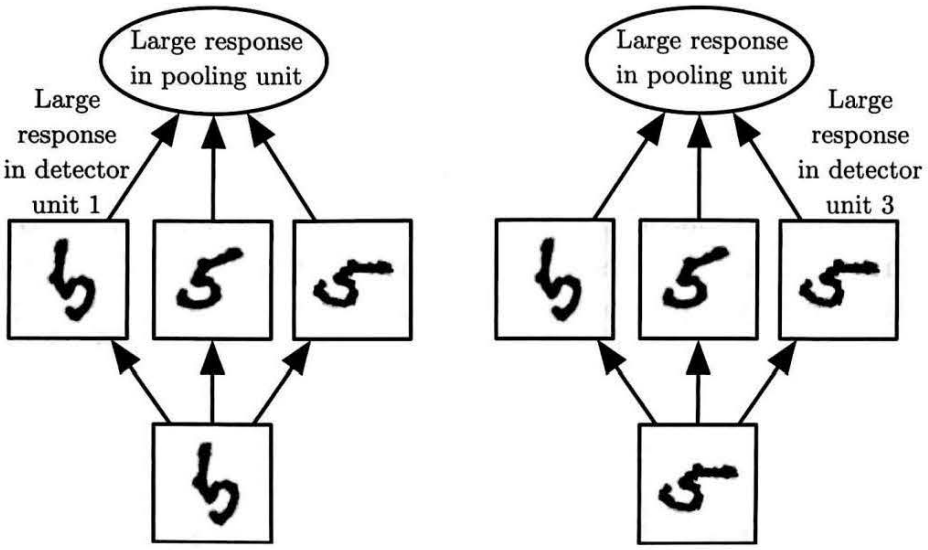


Figure 9.9: Example of learned invariances. A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand written 5. Each filter attempts to match a slightly different orientation of the 5. When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which detector unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way. This principle is leveraged by maxout networks (Goodfellow et al., 2013a) and other convolutional networks. Max pooling over spatial positions is naturally invariant to translation; this multichannel approach is only necessary for learning other transformations.

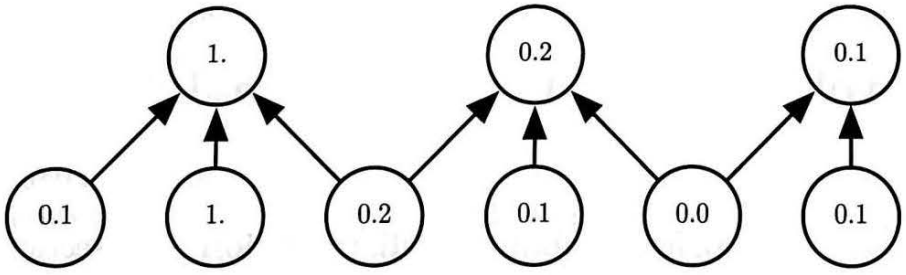


Figure 9.10: Pooling with downsampling. Here we use max pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer. Note that the rightmost pooling region has a smaller size but must be included if we do not want to ignore some of the detector units.

Microsoft Exhibit 1043 - Page 120
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

when the next layer is fully connected and based on matrix multiplication), this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau et al., 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau et al., 2011). This approach yields a different set of pooling regions for each image. Another approach is to *learn* a single pooling structure that is then applied to all images (Jia et al., 2012).

Pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders. These issues are discussed further when we present these types of networks in part III. Pooling in convolutional Boltzmann machines is presented in section 20.6. The inverse-like operations on pooling units needed in some differentiable networks are covered in section 20.10.6.

Some examples of complete convolutional network architectures for classification using convolution and pooling are shown in figure 9.11.

9.4 Convolution and Pooling as an Infinitely Strong Prior

Recall the concept of a **prior probability distribution** from section 5.2. This is a probability distribution over the parameters of a model that encodes our beliefs about what models are reasonable, before we have seen any data.

Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a prior distribution with high entropy, such as a Gaussian distribution with high variance. Such a prior allows the data to move the parameters more or less freely. A strong prior has very low

Microsoft Exhibit 1043 - Page 121
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

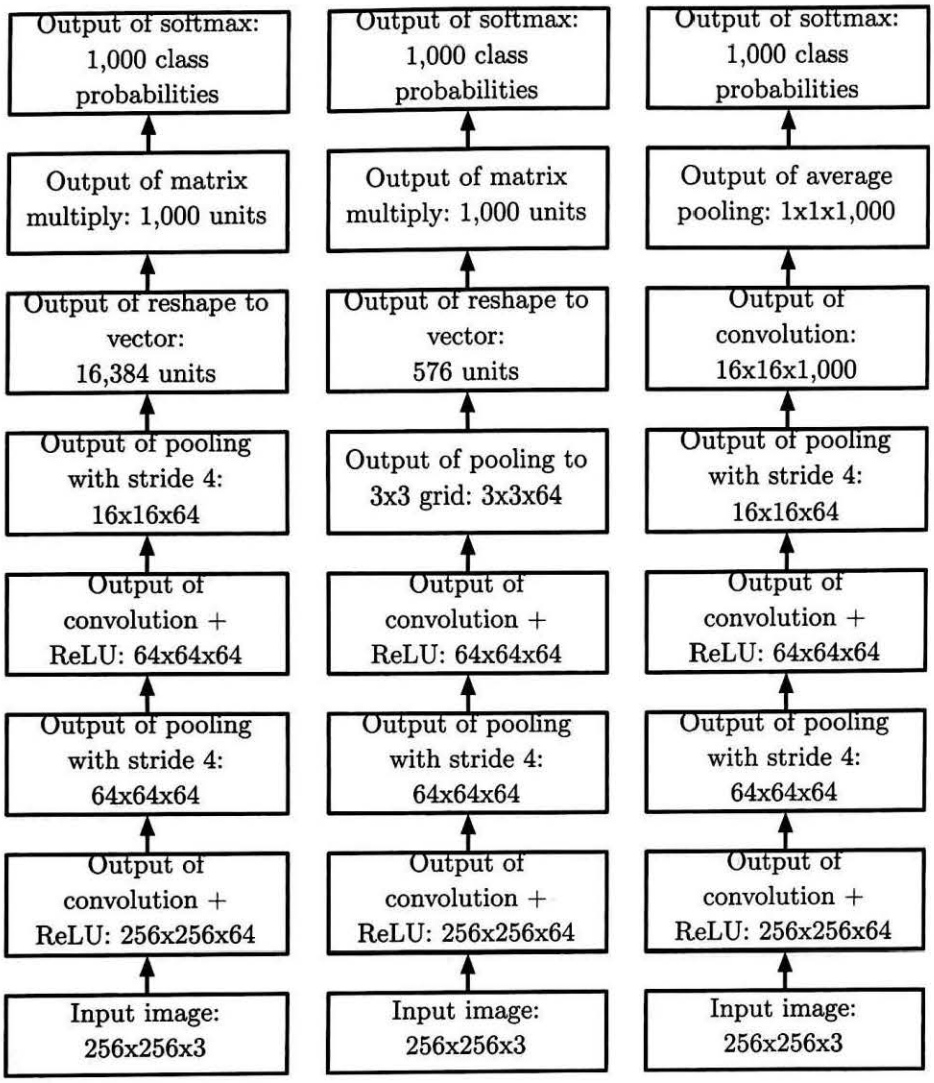


Figure 9.11: Examples of architectures for classification with convolutional networks. The specific strides and depths used in this figure are not advisable for real use; they are designed to be very shallow to fit onto the page. Real convolutional networks also often involve significant amounts of branching, unlike the chain structures used here for simplicity. *(Left)*A convolutional network that processes a fixed image size. After alternating between convolution and pooling for a few layers, the tensor for the convolutional feature map is reshaped to flatten out the spatial dimensions. The rest of the network is an ordinary feedforward network classifier, as described in chapter 6. *(Center)*A convolutional network that processes a variably sized image but still maintains a fully connected section. This network uses a pooling operation with variably sized pools but a fixed number of pools, in order to provide a fixed-size vector of 576 units to the fully connected portion of the network. *(Right)*A convolutional network that does not have any fully connected weight layer. Instead, the last convolutional layer outputs one feature map per class. The model presumably learns a map of how likely each class is to occur at each spatial location. Averaging a feature map down to a single value provides the argument to the softmax classifier at the top.

Microsoft Exhibit 1043 - Page 122
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

entropy, such as a Gaussian distribution with low variance. Such a prior plays a more active role in determining where the parameters end up.

An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data give to those values.

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit. Overall, we can think of the use of convolution as introducing an infinitely strong prior probability distribution over the parameters of a layer. This prior says that the function the layer should learn contains only local interactions and is equivariant to translation. Likewise, the use of pooling is an infinitely strong prior that each unit should be invariant to small translations.

Of course, implementing a convolutional net as a fully connected net with an infinitely strong prior would be extremely wasteful computationally. But thinking of a convolutional net as a fully connected net with an infinitely strong prior can give us some insights into how convolutional nets work.

One key insight is that convolution and pooling can cause underfitting. Like any prior, convolution and pooling are only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving precise spatial information, then using pooling on all features can increase the training error. Some convolutional network architectures (Szegedy et al., 2014a) are designed to use pooling on some channels but not on other channels, in order to get both highly invariant features and features that will not underfit when the translation invariance prior is incorrect. When a task involves incorporating information from very distant locations in the input, then the prior imposed by convolution may be inappropriate.

Another key insight from this view is that we should only compare convolutional models to other convolutional models in benchmarks of statistical learning performance. Models that do not use convolution would be able to learn even if we permuted all the pixels in the image. For many image datasets, there are separate benchmarks for models that are **permutation invariant** and must discover the concept of topology via learning and for models that have the knowledge of spatial relationships hard coded into them by their designer.

Microsoft Exhibit 1043 - Page 123
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

9.5 Variants of the Basic Convolution Function

When discussing convolution in the context of neural networks, we usually do not refer exactly to the standard discrete convolution operation as it is usually understood in the mathematical literature. The functions used in practice differ slightly. Here we describe these differences in detail and highlight some useful properties of the functions used in neural networks.

First, when we refer to convolution in the context of neural networks, we usually actually mean an operation that consists of many applications of convolution in parallel. This is because convolution with a single kernel can extract only one kind of feature, albeit at many spatial locations. Usually we want each layer of our network to extract many kinds of features, at many locations.

Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations. For example, a color image has a red, green and blue intensity at each pixel. In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel. Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch, but we will omit the batch axis in our description here for simplicity.

Because convolutional networks usually use multichannel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel flipping is used. These multichannel operations are only commutative if each operation has the same number of output channels as input channels.

Assume we have a 4-D kernel tensor $\mathbf{K}$ with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit. Assume our input consists of observed data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k. Assume our output consists of $\mathbf{Z}$ with the same format as $\mathbf{V}$. If $\mathbf{Z}$ is produced by convolving $\mathbf{K}$ across $\mathbf{V}$ without flipping $\mathbf{K}$, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}, \tag{9.7}$$

Microsoft Exhibit 1043 - Page 124
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

where the summation over l, m and n is over all values for which the tensor indexing operations inside the summation are valid. In linear algebra notation, we index into arrays using a 1 for the first entry. This necessitates the -1 in the above formula. Programming languages such as C and Python index starting from 0, rendering the above expression even simpler.

We may want to skip over some positions of the kernel to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as downsampling the output of the full convolution function. If we want to sample only every s pixels in each direction in the output, then we can define a downsampled convolution function c such that

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right]. \tag{9.8}$$

We refer to s as the **stride** of this downsampled convolution. It is also possible to define a separate stride for each direction of motion. See figure 9.12 for an illustration.

One essential feature of any convolutional network implementation is the ability to implicitly zero pad the input $\mathbf{V}$ to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels—both scenarios that significantly limit the expressive power of the network. See figure 9.13 for an example.

Three special cases of the zero-padding setting are worth mentioning. One is the extreme case in which no zero padding is used whatsoever, and the convolution kernel is allowed to visit only positions where the entire kernel is contained entirely within the image. In MATLAB terminology, this is called **valid** convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer. If the input image has width m and the kernel has width k, the output will be of width $m - k + 1$. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to 1×1, at which point additional layers cannot meaningfully be considered convolutional. Another special case of the zero-padding setting is when just enough zero padding is added to keep the size of the output equal to the size of the input. MATLAB calls this **same** convolution. In this case, the network

Microsoft Exhibit 1043 - Page 125
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

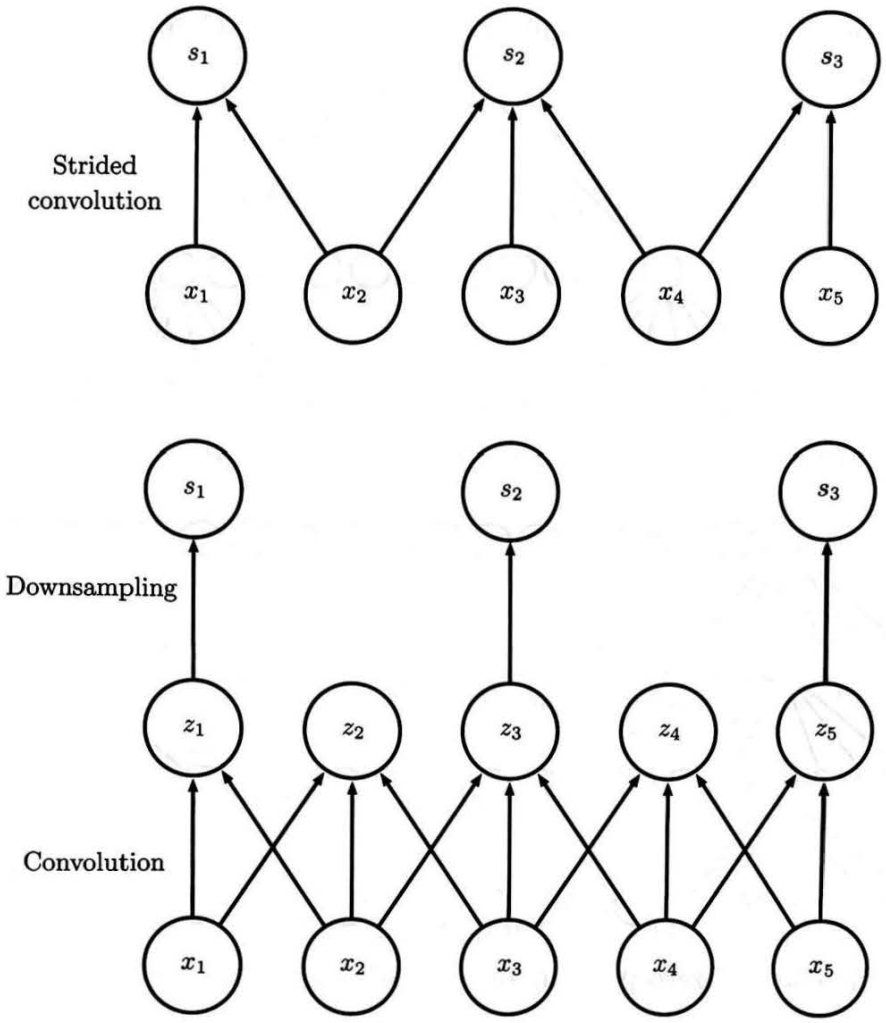


Figure 9.12: Convolution with a stride. In this example, we use a stride of two. *(Top)*Convolution with a stride length of two implemented in a single operation. *(Bottom)*Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by downsampling. Obviously, the two-step approach involving downsampling is computationally wasteful, because it computes many values that are then discarded.

can contain as many convolutional layers as the available hardware can support, since the operation of convolution does not modify the architectural possibilities available to the next layer. The input pixels near the border, however, influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model. This motivates the other extreme case, which MATLAB refers to as **full** convolution, in which enough zeros are added for every pixel to be visited k times in each direction, resulting in an output image of width $m + k - 1$. In this case, the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult

Microsoft Exhibit 1043 - Page 126
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

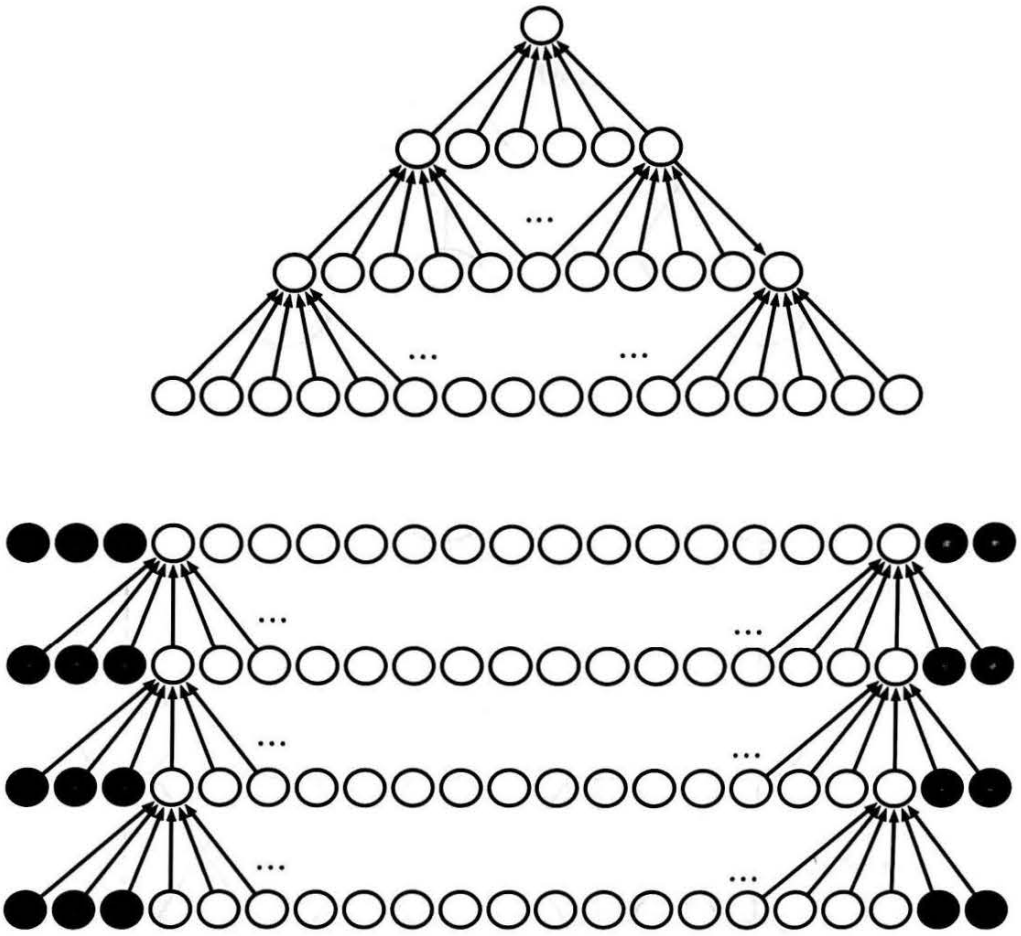


Figure 9.13: The effect of zero padding on network size. Consider a convolutional network with a kernel of width six at every layer. In this example, we do not use any pooling, so only the convolution operation itself shrinks the network size. *(Top)*In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive, and some shrinking is inevitable in this kind of architecture. *(Bottom)*By adding five implicit zeros to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding (in terms of test set classification accuracy) lies somewhere between "valid" and "same" convolution.

In some cases, we do not actually want to use convolution, but want to use locally connected layers instead (LeCun, 1986, 1989). In this case, the adjacency matrix in the graph of our MLP is the same, but every connection has its own

Microsoft Exhibit 1043 - Page 127
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

weight, specified by a 6-D tensor $\mathbf{W}$. The indices into $\mathbf{W}$ are respectively: i; the output channel; j, the output row; k, the output column; l, the input channel; m, the row offset within the input; and n, the column offset within the input. The linear part of a locally connected layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} \left[V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n} \right]. \tag{9.9}$$

This is sometimes also called **unshared convolution**, because it is a similar operation to discrete convolution with a small kernel, but without sharing parameters across locations. Figure 9.14 compares local connections, convolution, and full connections.

Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space. For example, if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image.

It can also be useful to make versions of convolution or locally connected layers in which the connectivity is further restricted, for example to constrain each output channel i to be a function of only a subset of the input channels l. A common way to do this is to make the first m output channels connect to only the first n input channels, the second m output channels connect to only the second n input channels, and so on. See figure 9.15 for an example. Modeling interactions between few channels allows the network to have fewer parameters, reducing memory consumption, increasing statistical efficiency, and reducing the amount of computation needed to perform forward and back-propagation. It accomplishes these goals without reducing the number of hidden units.

Tiled convolution (Gregor and LeCun, 2010a; Le et al., 2010) offers a compromise between a convolutional layer and a locally connected layer. Rather than learning a separate set of weights at *every* spatial location, we learn a set of kernels that we rotate through as we move through space. This means that immediately neighboring locations will have different filters, as in a locally connected layer, but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels, rather than by the size of the entire output feature map. See figure 9.16 for a comparison of locally connected layers, tiled convolution, and standard convolution.

To define tiled convolution algebraically, let k be a 6-D tensor, where two of the dimensions correspond to different locations in the output map. Rather than having a separate index for each location in the output map, output locations cycle

Microsoft Exhibit 1043 - Page 128
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

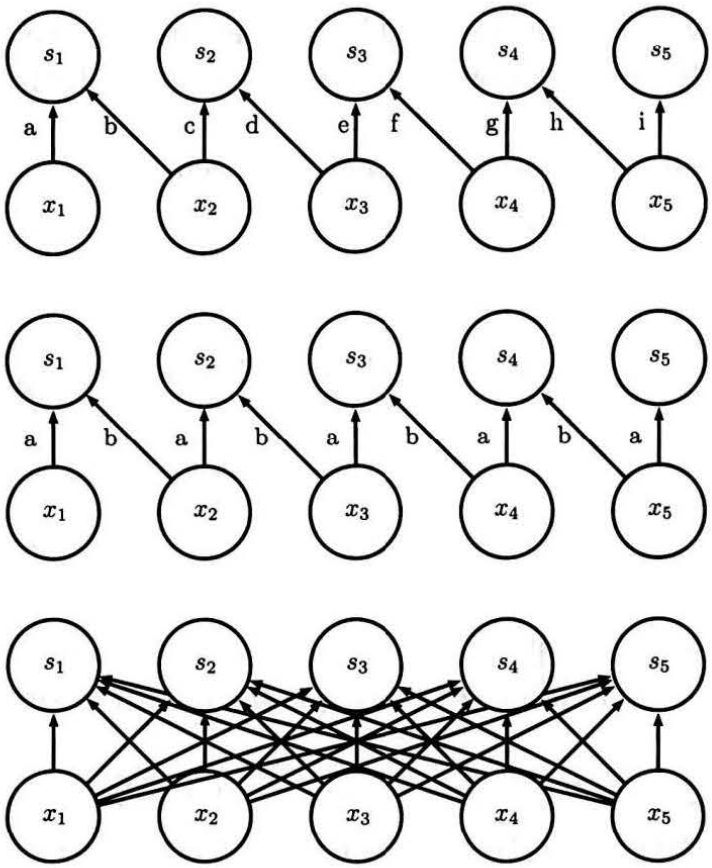


Figure 9.14: Comparison of local connections, convolution, and full connections. *(Top)*A locally connected layer with a patch size of two pixels. Each edge is labeled with a unique letter to show that each edge is associated with its own weight parameter. *(Center)*A convolutional layer with a kernel width of two pixels. This model has exactly the same connectivity as the locally connected layer. The difference lies not in which units interact with each other, but in how the parameters are shared. The locally connected layer has no parameter sharing. The convolutional layer uses the same two weights repeatedly across the entire input, as indicated by the repetition of the letters labeling each edge. *(Bottom)*A fully connected layer resembles a locally connected layer in the sense that each edge has its own parameter (there are too many to label explicitly with letters in this diagram). It does not, however, have the restricted connectivity of the locally connected layer.

through a set of t different choices of kernel stack in each direction. If t is equal to the output width, this is the same as a locally connected layer.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j\%t+1,k\%t+1}, \tag{9.10}$$

where percent is the modulo operation, with $t\%t = 0$, $(t+1)\%t = 1$, and so on. It is straightforward to generalize this equation to use a different tiling range for each dimension.

Microsoft Exhibit 1043 - Page 129
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

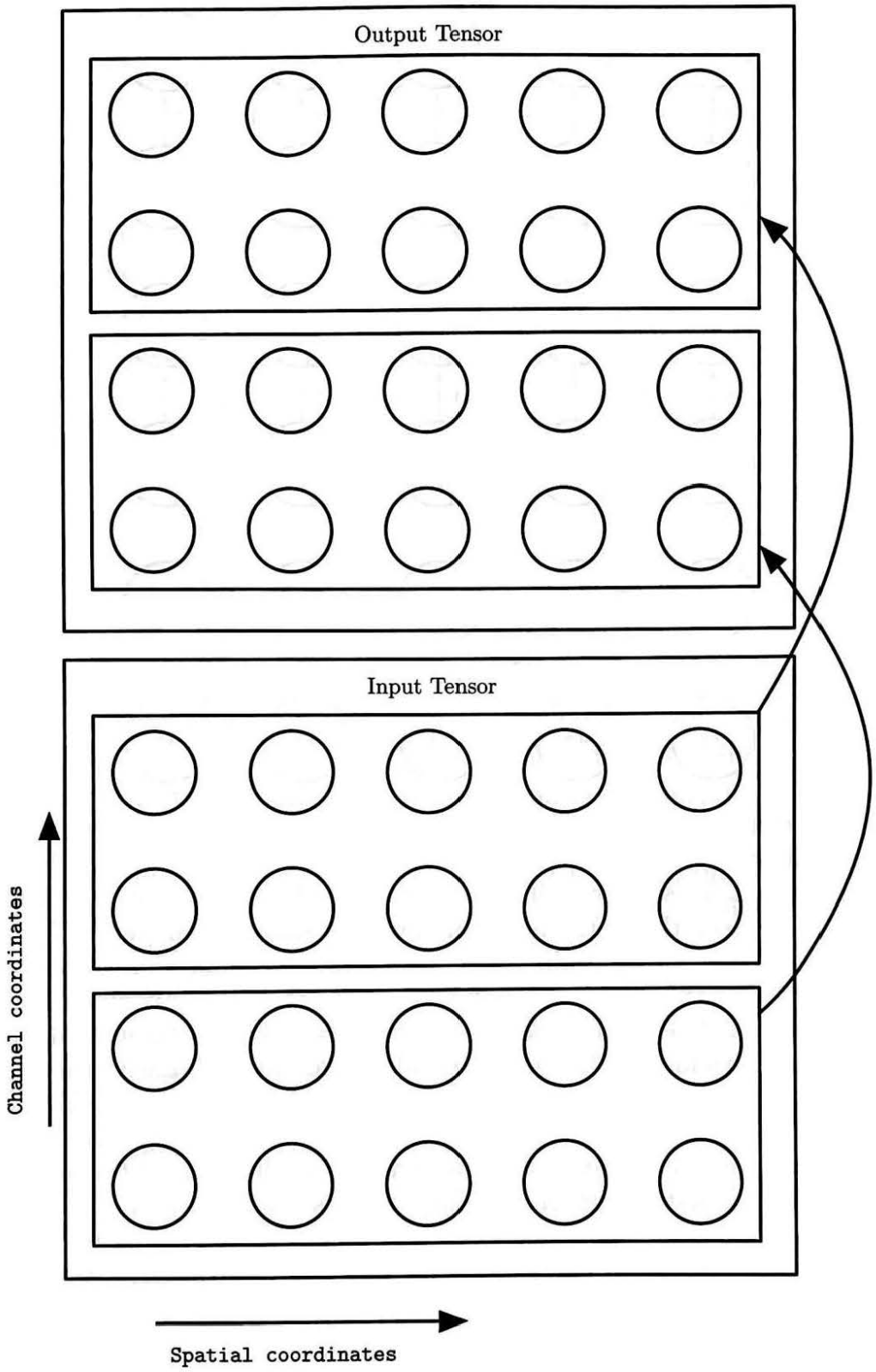


Figure 9.15: A convolutional network with the first two output channels connected to only the first two input channels, and the second two output channels connected to only the second two input channels.

Microsoft Exhibit 1043 - Page 130
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

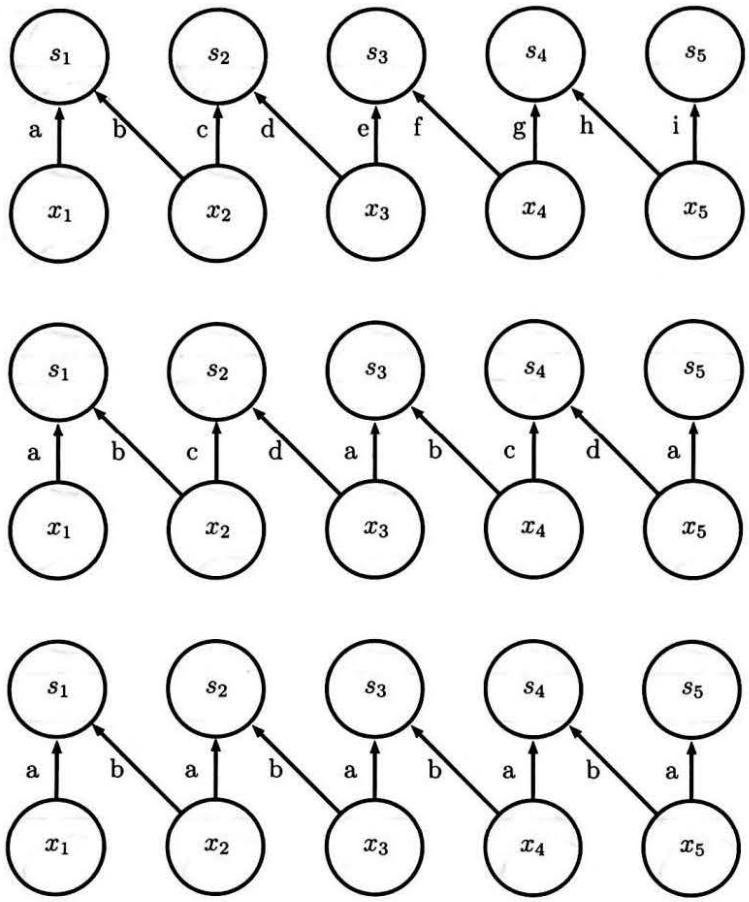


Figure 9.16: A comparison of locally connected layers, tiled convolution, and standard convolution. All three have the same sets of connections between units, when the same size of kernel is used. This diagram illustrates the use of a kernel that is two pixels wide. The differences between the methods lies in how they share parameters. *(Top)* A locally connected layer has no sharing at all. We indicate that each connection has its own weight by labeling each connection with a unique letter. *(Center)* Tiled convolution has a set of t different kernels. Here we illustrate the case of $t = 2$. One of these kernels has edges labeled "a" and "b," while the other has edges labeled "c" and "d." Each time we move one pixel to the right in the output, we move on to using a different kernel. This means that, like the locally connected layer, neighboring units in the output have different parameters. Unlike the locally connected layer, after we have gone through all t available kernels, we cycle back to the first kernel. If two output units are separated by a multiple of t steps, then they share parameters. *(Bottom)* Traditional convolution is equivalent to tiled convolution with $t = 1$. There is only one kernel, and it is applied everywhere, as indicated in the diagram by using the kernel with weights labeled "a" and "b" everywhere.

Locally connected layers and tiled convolutional layers have both an interesting interaction with max pooling: the detector units of these layers are driven by different filters. If these filters learn to detect different transformed versions of the same underlying features, then the max-pooled units become invariant to the

Microsoft Exhibit 1043 - Page 131
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

learned transformation (see figure 9.9). Convolutional layers are hard coded to be invariant specifically to translation.

Other operations besides convolution are usually necessary to implement a convolutional network. To perform learning, one must be able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs. In some simple cases, this operation can be performed using the convolution operation, but many cases of interest, including the case of stride greater than 1, do not have this property.

Recall that convolution is a linear operation and can thus be described as a matrix multiplication (if we first reshape the input tensor into a flat vector). The matrix involved is a function of the convolution kernel. The matrix is sparse, and each element of the kernel is copied to several elements of the matrix. This view helps us to derive some of the other operations needed to implement a convolutional network.

Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to back-propagate error derivatives through a convolutional layer, so it is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed if we wish to reconstruct the visible units from the hidden units (Simard et al., 1992). Reconstructing the visible units is an operation commonly used in the models described in part III of this book, such as autoencoders, RBMs, and sparse coding. Transpose convolution is necessary to construct convolutional versions of those models. Like the kernel gradient operation, this input gradient operation can sometimes be implemented using a convolution but in general requires a third operation to be implemented. Care must be taken to coordinate this transpose operation with the forward propagation. The size of the output that the transpose operation should return depends on the zero-padding policy and stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

These three operations—convolution, backprop from output to weights, and backprop from output to inputs—are sufficient to compute all the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution. See Goodfellow (2010) for a full derivation of the equations in the fully general multidimensional, multiexample case. To give a sense of how these equations work, we present the two-dimensional, single example version here.

Microsoft Exhibit 1043 - Page 132
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

Suppose we want to train a convolutional network that incorporates strided convolution of kernel stack $\mathbf{K}$ applied to multichannel image $\mathbf{V}$ with stride s as defined by $c(\mathbf{K}, \mathbf{V}, s)$, as in equation 9.8. Suppose we want to minimize some loss function $J(\mathbf{V}, \mathbf{K})$. During forward propagation, we will need to use c itself to output $\mathbf{Z}$, which is then propagated through the rest of the network and used to compute the cost function J. During back-propagation, we will receive a tensor $\mathbf{G}$ such that $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$.

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k,(n-1)\times s+l}. \tag{9.11}$$

If this layer is not the bottom layer of the network, we will need to compute the gradient with respect to $\mathbf{V}$ to back-propagate the error further down. To do so, we can use a function

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) \tag{9.12}$$

$$= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)\times s+p=k}} \sum_{q} K_{q,i,m,p} G_{q,l,n}. \tag{9.13}$$

Autoencoder networks, described in chapter 14, are feedforward networks trained to copy their input to their output. A simple example is the PCA algorithm, which copies its input $\boldsymbol{x}$ to an approximate reconstruction $\boldsymbol{r}$ using the function $\boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{x}$. It is common for more general autoencoders to use multiplication by the transpose of the weight matrix just as PCA does. To make such models convolutional, we can use the function h to perform the transpose of the convolution operation. Suppose we have hidden units $\mathbf{H}$ in the same format as $\mathbf{Z}$ and we define a reconstruction

$$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s). \tag{9.14}$$

To train the autoencoder, we will receive the gradient with respect to $\mathbf{R}$ as a tensor $\mathbf{E}$. To train the decoder, we need to obtain the gradient with respect to $\mathbf{K}$. This is given by $g(\mathbf{H}, \mathbf{E}, s)$. To train the encoder, we need to obtain the gradient with respect to $\mathbf{H}$. This is given by $c(\mathbf{K}, \mathbf{E}, s)$. It is also possible to differentiate through g using c and h, but these operations are not needed for the back-propagation algorithm on any standard network architectures.

Microsoft Exhibit 1043 - Page 133
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

Generally, we do not use only a linear operation to transform from the inputs to the outputs in a convolutional layer. We generally also add some bias term to each output before applying the nonlinearity. This raises the question of how to share parameters among the biases. For locally connected layers, it is natural to give each unit its own bias, and for tiled convolution, it is natural to share the biases with the same tiling pattern as the kernels. For convolutional layers, it is typical to have one bias per channel of the output and share it across all locations within each convolution map. If the input is of known, fixed size, however, it is also possible to learn a separate bias at each location of the output map. Separating the biases may slightly reduce the statistical efficiency of the model, but it allows the model to correct for differences in the image statistics at different locations. For example, when using implicit zero padding, detector units at the edge of the image receive less total input and may need larger biases.

9.6 Structured Outputs

Convolutional networks can be used to output a high-dimensional structured object, rather than just predicting a class label for a classification task or a real value for a regression task. Typically this object is just a tensor, emitted by a standard convolutional layer. For example, the model might emit a tensor $\mathbf{S}$, where $S_{i,j,k}$ is the probability that pixel (j, k) of the input to the network belongs to class i. This allows the model to label every pixel in an image and draw precise masks that follow the outlines of individual objects.

One issue that often comes up is that the output plane can be smaller than the input plane, as shown in figure 9.13. In the kinds of architectures typically used for classification of a single object in an image, the greatest reduction in the spatial dimensions of the network comes from using pooling layers with large stride. To produce an output map of similar size as the input, one can avoid pooling altogether (Jain et al., 2007). Another strategy is to simply emit a lower-resolution grid of labels (Pinheiro and Collobert, 2014, 2015). Finally, in principle, one could use a pooling operator with unit stride.

One strategy for pixel-wise labeling of images is to produce an initial guess of the image labels, then refine this initial guess using the interactions between neighboring pixels. Repeating this refinement step several times corresponds to using the same convolutions at each stage, sharing weights between the last layers of the deep net (Jain et al., 2007). This makes the sequence of computations performed by the successive convolutional layers with weights shared across layers a particular kind of recurrent network (Pinheiro and Collobert, 2014, 2015). Figure 9.17 shows the architecture of such a recurrent convolutional network.

Microsoft Exhibit 1043 - Page 134
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

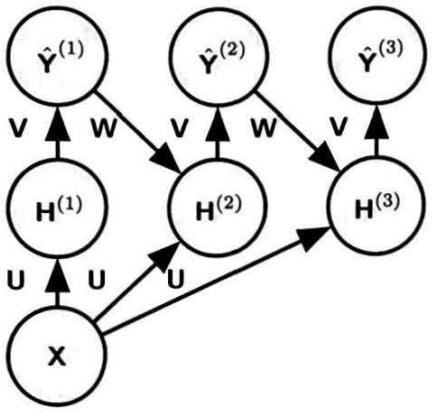


Figure 9.17: An example of a recurrent convolutional network for pixel labeling. The input is an image tensor $\mathbf{X}$, with axes corresponding to image rows, image columns, and channels (red, green, blue). The goal is to output a tensor of labels $\hat{Y}$, with a probability distribution over labels for each pixel. This tensor has axes corresponding to image rows, image columns, and the different classes. Rather than outputting $\hat{Y}$ in a single shot, the recurrent network iteratively refines its estimate $\hat{Y}$ by using a previous estimate of $\hat{Y}$ as input for creating a new estimate. The same parameters are used for each updated estimate, and the estimate can be refined as many times as we wish. The tensor of convolution kernels $\mathbf{U}$ is used on each step to compute the hidden representation given the input image. The kernel tensor $\mathbf{V}$ is used to produce an estimate of the labels given the hidden values. On all but the first step, the kernels $\mathbf{W}$ are convolved over $\hat{Y}$ to provide input to the hidden layer. On the first time step, this term is replaced by zero. Because the same parameters are used on each step, this is an example of a recurrent network, as described in chapter 10.

Once a prediction for each pixel is made, various methods can be used to further process these predictions to obtain a segmentation of the image into regions (Briggman et al., 2009; Turaga et al., 2010; Farabet et al., 2013). The general idea is to assume that large groups of contiguous pixels tend to be associated with the same label. Graphical models can describe the probabilistic relationships between neighboring pixels. Alternatively, the convolutional network can be trained to maximize an approximation of the graphical model training objective (Ning et al., 2005; Thompson et al., 2014).

9.7 Data Types

The data used with a convolutional network usually consist of several channels, each channel being the observation of a different quantity at some point in space or time. See table 9.1 for examples of data types with different dimensionalities and number of channels.

Microsoft Exhibit 1043 - Page 135
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

	Single channel	Multichannel
1-D	Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step.	Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a "skeleton" over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character's skeleton. Each channel in the data we feed to the convolutional model represents the angle about one axis of one joint.
2-D	Audio data that has been preprocessed with a Fourier transform: We can transform the audio waveform into a 2-D tensor with different rows corresponding to different frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time. Using convolution across the frequency axis makes the model equivariant to frequency, so that the same melody played in a different octave produces the same representation but at a different height in the network's output.	Color image data: One channel contains the red pixels, one the green pixels, and one the blue pixels. The convolution kernel moves over both the horizontal and the vertical axes of the image, conferring translation equivariance in both directions.
3-D	Volumetric data: A common source of this kind of data is medical imaging technology, such as CT scans.	Color video data: One axis corresponds to time, one to the height of the video frame, and one to the width of the video frame.

Table 9.1: Examples of different formats of data that can be used with convolutional networks.

Microsoft Exhibit 1043 - Page 136
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

For an example of convolutional networks applied to video, see Chen et al. (2010).

So far we have discussed only the case where every example in the train and test data has the same spatial dimensions. One advantage to convolutional networks is that they can also process inputs with varying spatial extents. These kinds of input simply cannot be represented by traditional, matrix multiplication-based neural networks. This provides a compelling reason to use convolutional networks even when computational cost and overfitting are not significant issues.

For example, consider a collection of images in which each image has a different width and height. It is unclear how to model such inputs with a weight matrix of fixed size. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly. Convolution may be viewed as matrix multiplication; the same convolution kernel induces a different size of doubly block circulant matrix for each size of input. Sometimes the output of the network as well as the input is allowed to have variable size, for example, if we want to assign a class label to each pixel of the input. In this case, no further design work is necessary. In other cases, the network must produce some fixed-size output, for example, if we want to assign a single class label to the entire image. In this case, we must make some additional design steps, like inserting a pooling layer whose pooling regions scale in size proportional to the size of the input, to maintain a fixed number of pooled outputs. Some examples of this kind of strategy are shown in figure 9.11.

Note that the use of convolution for processing variably sized inputs makes sense only for inputs that have variable size because they contain varying amounts of observation of the same kind of thing—different lengths of recordings over time, different widths of observations over space, and so forth. Convolution does not make sense if the input has variable size because it can optionally include different kinds of observations. For example, if we are processing college applications, and our features consist of both grades and standardized test scores, but not every applicant took the standardized test, then it does not make sense to convolve the same weights over features corresponding to the grades as well as the features corresponding to the test scores.

9.8 Efficient Convolution Algorithms

Modern convolutional network applications often involve networks containing more than one million units. Powerful implementations exploiting parallel computation

Microsoft Exhibit 1043 - Page 137
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

resources, as discussed in section 12.1, are essential. In many cases, however, it is also possible to speed up convolution by selecting an appropriate convolution algorithm.

Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform. For some problem sizes, this can be faster than the naive implementation of discrete convolution.

When a d-dimensional kernel can be expressed as the outer product of d vectors, one vector per dimension, the kernel is called **separable**. When the kernel is separable, naive convolution is inefficient. It is equivalent to compose d one-dimensional convolutions with each of these vectors. The composed approach is significantly faster than performing one d-dimensional convolution with their outer product. The kernel also takes fewer parameters to represent as vectors. If the kernel is w elements wide in each dimension, then naive multidimensional convolution requires $O(w^d)$ runtime and parameter storage space, while separable convolution requires $O(w \times d)$ runtime and parameter storage space. Of course, not every convolution can be represented in this way.

Devising faster ways of performing convolution or approximate convolution without harming the accuracy of the model is an active area of research. Even techniques that improve the efficiency of only forward propagation are useful because in the commercial setting, it is typical to devote more resources to deployment of a network than to its training.

9.9 Random or Unsupervised Features

Typically, the most expensive part of convolutional network training is learning the features. The output layer is usually relatively inexpensive because of the small number of features provided as input to this layer after passing through several layers of pooling. When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation through the entire network. One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.

There are three basic strategies for obtaining convolution kernels without supervised training. One is to simply initialize them randomly. Another is to design them by hand, for example, by setting each kernel to detect edges at a certain orientation or scale. Finally, one can learn the kernels with an unsupervised criterion. For example, Coates et al. (2011) apply k-means clustering to

Microsoft Exhibit 1043 - Page 138
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

small image patches, then use each learned centroid as a convolution kernel. In Part III we describe many more unsupervised learning approaches. Learning the features with an unsupervised criterion allows them to be determined separately from the classifier layer at the top of the architecture. One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer. Learning the last layer is then typically a convex optimization problem, assuming the last layer is something like logistic regression or an SVM.

Random filters often work surprisingly well in convolutional networks (Jarrett et al., 2009; Saxe et al., 2011; Pinto et al., 2011; Cox and Pinto, 2011). Saxe et al. (2011) showed that layers consisting of convolution followed by pooling naturally become frequency selective and translation invariant when assigned random weights. They argue that this provides an inexpensive way to choose the architecture of a convolutional network: first, evaluate the performance of several convolutional network architectures by training only the last layer; then take the best of these architectures and train the entire architecture using a more expensive approach.

An intermediate approach is to learn the features, but using methods that do not require full forward and back-propagation at every gradient step. As with multilayer perceptrons, we use greedy layer-wise pretraining, to train the first layer in isolation, then extract all features from the first layer only once, then train the second layer in isolation given those features, and so on. In chapter 8 we described how to perform supervised greedy layer-wise pretraining, and in part III extend this to greedy layer-wise pretraining using an unsupervised criterion at each layer. The canonical example of greedy layer-wise pretraining of a convolutional model is the convolutional deep belief network (Lee et al., 2009). Convolutional networks offer us the opportunity to take the pretraining strategy one step further than is possible with multilayer perceptrons. Instead of training an entire convolutional layer at a time, we can train a model of a small patch, as Coates et al. (2011) do with k-means. We can then use the parameters from this patch-based model to define the kernels of a convolutional layer. This means that it is possible to use unsupervised learning to train a convolutional network *without ever using convolution during the training process.* Using this approach, we can train very large models and incur a high computational cost only at inference time (Ranzato et al., 2007b; Jarrett et al., 2009; Kavukcuoglu et al., 2010; Coates et al., 2013). This approach was popular from roughly 2007 to 2013, when labeled datasets were small and computational power was more limited. Today, most convolutional networks are trained in a purely supervised fashion, using full forward and back-propagation through the entire network on each training iteration.

Microsoft Exhibit 1043 - Page 139
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

As with other approaches to unsupervised pretraining, it remains difficult to tease apart the cause of some of the benefits seen with this approach. Unsupervised pretraining may offer some regularization relative to supervised training, or it may simply allow us to train much larger architectures because of the reduced computational cost of the learning rule.

9.10 The Neuroscientific Basis for Convolutional Networks

Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence. Though convolutional networks have been guided by many other fields, some of the key design principles of neural networks were drawn from neuroscience.

The history of convolutional networks begins with neuroscientific experiments long before the relevant computational models were developed. Neurophysiologists David Hubel and Torsten Wiesel collaborated for several years to determine many of the most basic facts about how the mammalian vision system works (Hubel and Wiesel, 1959, 1962, 1968). Their accomplishments were eventually recognized with a Nobel prize. Their findings that have had the greatest influence on contemporary deep learning models were based on recording the activity of individual neurons in cats. They observed how neurons in the cat's brain responded to images projected in precise locations on a screen in front of the cat. Their great discovery was that neurons in the early visual system responded most strongly to very specific patterns of light, such as precisely oriented bars, but responded hardly at all to other patterns.

Their work helped to characterize many aspects of brain function that are beyond the scope of this book. From the point of view of deep learning, we can focus on a simplified, cartoon view of brain function.

In this simplified view, we focus on a part of the brain called V1, also known as the **primary visual cortex**. V1 is the first area of the brain that begins to perform significantly advanced processing of visual input. In this cartoon view, images are formed by light arriving in the eye and stimulating the retina, the light-sensitive tissue in the back of the eye. The neurons in the retina perform some simple preprocessing of the image but do not substantially alter the way it is represented. The image then passes through the optic nerve and a brain region called the *lateral geniculate nucleus*. The main role, as far as we are concerned here, of both anatomical regions is primarily just to carry the signal from the eye to V1, which is located at the back of the head.

Microsoft Exhibit 1043 - Page 140
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

A convolutional network layer is designed to capture three properties of V1:

1. V1 is arranged in a spatial map. It actually has a two-dimensional structure, mirroring the structure of the image in the retina. For example, light arriving at the lower half of the retina affects only the corresponding half of V1. Convolutional networks capture this property by having their features defined in terms of two-dimensional maps.

2. V1 contains many **simple cells**. A simple cell's activity can to some extent be characterized by a linear function of the image in a small, spatially localized receptive field. The detector units of a convolutional network are designed to emulate these properties of simple cells.

3. V1 also contains many **complex cells**. These cells respond to features that are similar to those detected by simple cells, but complex cells are invariant to small shifts in the position of the feature. This inspires the pooling units of convolutional networks. Complex cells are also invariant to some changes in lighting that cannot be captured simply by pooling over spatial locations. These invariances have inspired some of the cross-channel pooling strategies in convolutional networks, such as maxout units (Goodfellow et al., 2013a).

Though we know the most about V1, it is generally believed that the same basic principles apply to other areas of the visual system. In our cartoon view of the visual system, the basic strategy of detection followed by pooling is repeatedly applied as we move deeper into the brain. As we pass through multiple anatomical layers of the brain, we eventually find cells that respond to some specific concept and are invariant to many transformations of the input. These cells have been nicknamed "grandmother cells"—the idea is that a person could have a neuron that activates when seeing an image of their grandmother, regardless of whether she appears in the left or right side of the image, whether the image is a close-up of her face or zoomed-out shot of her entire body, whether she is brightly lit or in shadow, and so on.

These grandmother cells have been shown to actually exist in the human brain, in a region called the *medial temporal lobe* (Quiroga et al., 2005). Researchers tested whether individual neurons would respond to photos of famous individuals. They found what has come to be called the "Halle Berry neuron," an individual neuron that is activated by the concept of Halle Berry. This neuron fires when a person sees a photo of Halle Berry, a drawing of Halle Berry, or even text containing the words "Halle Berry." Of course, this has nothing to do with Halle Berry herself;

Microsoft Exhibit 1043 - Page 141
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

other neurons responded to the presence of Bill Clinton, Jennifer Aniston, and so forth.

These medial temporal lobe neurons are somewhat more general than modern convolutional networks, which would not automatically generalize to identifying a person or object when reading its name. The closest analog to a convolutional network's last layer of features is a brain area called the *inferotemporal cortex* (IT). When viewing an object, information flows from the retina, through the LGN, to V1, then onward to V2, then V4, then IT. This happens within the first 100ms of glimpsing an object. If a person is allowed to continue looking at the object for more time, then information will begin to flow backward as the brain uses top-down feedback to update the activations in the lower level brain areas. If we interrupt the person's gaze, however, and observe only the firing rates that result from the first 100ms of mostly feedforward activation, then IT proves to be similar to a convolutional network. Convolutional networks can predict IT firing rates and perform similarly to (time-limited) humans on object recognition tasks (DiCarlo, 2013).

That being said, there are many differences between convolutional networks and the mammalian vision system. Some of these differences are well known to computational neuroscientists but outside the scope of this book. Some of these differences are not yet known, because many basic questions about how the mammalian vision system works remain unanswered. As a brief list:

- The human eye is mostly very low resolution, except for a tiny patch called the **fovea**. The fovea only observes an area about the size of a thumbnail held at arms length. Though we feel as if we can see an entire scene in high resolution, this is an illusion created by the subconscious part of our brain, as it stitches together several glimpses of small areas. Most convolutional networks actually receive large full-resolution photographs as input. The human brain makes several eye movements called **saccades** to glimpse the most visually salient or task-relevant parts of a scene. Incorporating similar attention mechanisms into deep learning models is an active research direction. In the context of deep learning, attention mechanisms have been most successful for natural language processing, as described in section 12.4.5.1. Several visual models with foveation mechanisms have been developed but so far have not become the dominant approach (Larochelle and Hinton, 2010; Denil et al., 2012).

- The human visual system is integrated with many other senses, such as hearing, and factors like our moods and thoughts. Convolutional networks so far are purely visual.

Microsoft Exhibit 1043 - Page 142
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

- The human visual system does much more than just recognize objects. It is able to understand entire scenes, including many objects and relationships between objects, and it processes rich 3-D geometric information needed for our bodies to interface with the world. Convolutional networks have been applied to some of these problems, but these applications are in their infancy.

- Even simple brain areas like V1 are heavily affected by feedback from higher levels. Feedback has been explored extensively in neural network models but has not yet been shown to offer a compelling improvement.

- While feedforward IT firing rates capture much of the same information as convolutional network features, it is not clear how similar the intermediate computations are. The brain probably uses very different activation and pooling functions. An individual neuron's activation probably is not well characterized by a single linear filter response. A recent model of V1 involves multiple quadratic filters for each neuron (Rust et al., 2005). Indeed our cartoon picture of "simple cells" and "complex cells" might create a nonexistent distinction; simple cells and complex cells might both be the same kind of cell but with their "parameters" enabling a continuum of behaviors ranging from what we call "simple" to what we call "complex."

It is also worth mentioning that neuroscience has told us relatively little about how to *train* convolutional networks. Model structures with parameter sharing across multiple spatial locations date back to early connectionist models of vision (Marr and Poggio, 1976), but these models did not use the modern back-propagation algorithm and gradient descent. For example, the neocognitron (Fukushima, 1980) incorporated most of the model architecture design elements of the modern convolutional network but relied on a layer-wise unsupervised clustering algorithm.

Lang and Hinton (1988) introduced the use of back-propagation to train **time-delay neural networks** (TDNNs). To use contemporary terminology, TDNNs are one-dimensional convolutional networks applied to time series. Back-propagation applied to these models was not inspired by any neuroscientific observation and is considered by some to be biologically implausible. Following the success of back-propagation-based training of TDNNs, LeCun et al. (1989) developed the modern convolutional network by applying the same training algorithm to 2-D convolution applied to images.

So far we have described how simple cells are roughly linear and selective for certain features, complex cells are more nonlinear and become invariant to some transformations of these simple cell features, and stacks of layers that alternate

Microsoft Exhibit 1043 - Page 143
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

between selectivity and invariance can yield grandmother cells for specific phenomena. We have not yet described precisely what these individual cells detect. In a deep nonlinear network, it can be difficult to understand the function of individual cells. Simple cells in the first layer are easier to analyze, because their responses are driven by a linear function. In an artificial neural network, we can just display an image of the convolution kernel to see what the corresponding channel of a convolutional layer responds to. In a biological neural network, we do not have access to the weights themselves. Instead, we put an electrode in the neuron, display several samples of white noise images in front of the animal's retina, and record how each of these samples causes the neuron to activate. We can then fit a linear model to these responses to obtain an approximation of the neuron's weights. This approach is known as **reverse correlation** (Ringach and Shapley, 2004).

Reverse correlation shows us that most V1 cells have weights that are described by **Gabor functions**. The Gabor function describes the weight at a 2-D point in the image. We can think of an image as being a function of 2-D coordinates, $I(x, y)$. Likewise, we can think of a simple cell as sampling the image at a set of locations, defined by a set of x coordinates $\mathbb{X}$ and a set of y coordinates $\mathbb{Y}$, then applying weights that are also a function of the location, $w(x, y)$. From this point of view, the response of a simple cell to an image is given by

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x, y) I(x, y). \tag{9.15}$$

Specifically, $w(x, y)$ takes the form of a Gabor function:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(f x' + \phi), \tag{9.16}$$

where

$$x' = (x - x_0)\cos(\tau) + (y - y_0)\sin(\tau) \tag{9.17}$$

and

$$y' = -(x - x_0)\sin(\tau) + (y - y_0)\cos(\tau). \tag{9.18}$$

Here, α, β_x, β_y, f, ϕ, x_0, y_0, and τ are parameters that control the properties of the Gabor function. Figure 9.18 shows some examples of Gabor functions with different settings of these parameters.

The parameters x_0, y_0, and τ define a coordinate system. We translate and rotate x and y to form x' and y'. Specifically, the simple cell will respond to image features centered at the point (x_0, y_0), and it will respond to changes in brightness as we move along a line rotated τ radians from the horizontal.

Microsoft Exhibit 1043 - Page 144
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

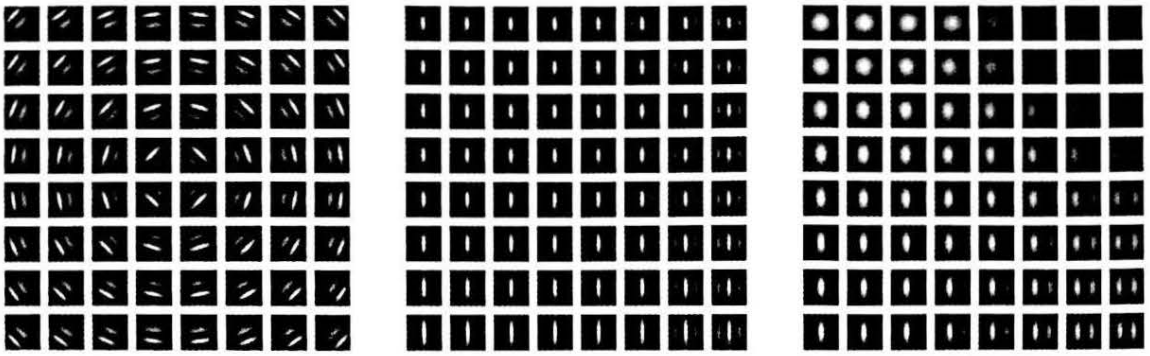


Figure 9.18: Gabor functions with a variety of parameter settings. White indicates large positive weight, black indicates large negative weight, and the background gray corresponds to zero weight. *(Left)*Gabor functions with different values of the parameters that control the coordinate system: x_0, y_0, and τ. Each Gabor function in this grid is assigned a value of x_0 and y_0 proportional to its position in its grid, and τ is chosen so that each Gabor filter is sensitive to the direction radiating out from the center of the grid. For the other two plots, x_0, y_0, and τ are fixed to zero. *(Center)*Gabor functions with different Gaussian scale parameters β_x and β_y. Gabor functions are arranged in increasing width (decreasing β_x) as we move left to right through the grid, and increasing height (decreasing β_y) as we move top to bottom. For the other two plots, the β values are fixed to 1.5 times the image width. *(Right)*Gabor functions with different sinusoid parameters f and ϕ. As we move top to bottom, f increases, and as we move left to right, ϕ increases. For the other two plots, ϕ is fixed to 0 and f is fixed to 5 times the image width.

Viewed as a function of x' and y', the function w then responds to changes in brightness as we move along the x' axis. It has two important factors: one is a Gaussian function, and the other is a cosine function.

The Gaussian factor $\alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right)$ can be seen as a gating term that ensures that the simple cell will respond only to values near where x' and y' are both zero, in other words, near the center of the cell's receptive field. The scaling factor α adjusts the total magnitude of the simple cell's response, while β_x and β_y control how quickly its receptive field falls off.

The cosine factor $\cos(fx' + \phi)$ controls how the simple cell responds to changing brightness along the x' axis. The parameter f controls the frequency of the cosine, and ϕ controls its phase offset.

Altogether, this cartoon view of simple cells means that a simple cell responds to a specific spatial frequency of brightness in a specific direction at a specific location. Simple cells are most excited when the wave of brightness in the image has the same phase as the weights. This occurs when the image is bright where the weights are positive and dark where the weights are negative. Simple cells are most

Microsoft Exhibit 1043 - Page 145
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

inhibited when the wave of brightness is fully out of phase with the weights—when the image is dark where the weights are positive and bright where the weights are negative.

The cartoon view of a complex cell is that it computes the L^2 norm of the 2-D vector containing two simple cells' responses: $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. An important special case occurs when s_1 has all the same parameters as s_0 except for ϕ, and ϕ is set such that s_1 is one quarter cycle out of phase with s_0. In this case, s_0 and s_1 form a **quadrature pair**. A complex cell defined in this way responds when the Gaussian reweighted image $I(x, y) \exp(-\beta_x x'^2 - \beta_y y'^2)$ contains a high-amplitude sinusoidal wave with frequency f in direction τ near (x_0, y_0), *regardless of the phase offset of this wave.* In other words, the complex cell is invariant to small translations of the image in direction τ, or to negating the image (replacing black with white and vice versa).

Some of the most striking correspondences between neuroscience and machine learning come from visually comparing the features learned by machine learning models with those employed by V1. Olshausen and Field (1996) showed that a simple unsupervised learning algorithm, sparse coding, learns features with receptive fields similar to those of simple cells. Since then, we have found that an extremely wide variety of statistical learning algorithms learn features with Gabor-like functions when applied to natural images. This includes most deep learning algorithms, which learn these features in their first layer. Figure 9.19 shows some examples. Because so many different learning algorithms learn edge detectors, it is difficult to conclude that any specific learning algorithm is the "right" model of the brain just based on the features it learns (though it can certainly be a bad sign if an algorithm does *not* learn some sort of edge detector when applied to natural images). These features are an important part of the statistical structure of natural images and can be recovered by many different approaches to statistical modeling. See Hyvärinen et al. (2009) for a review of the field of natural image statistics.

9.11 Convolutional Networks and the History of Deep Learning

Convolutional networks have played an important role in the history of deep learning. They are a key example of a successful application of insights obtained by studying the brain to machine learning applications. They were also some of the first deep models to perform well, long before arbitrary deep models were considered viable. Convolutional networks were also some of the first neural

Microsoft Exhibit 1043 - Page 146
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

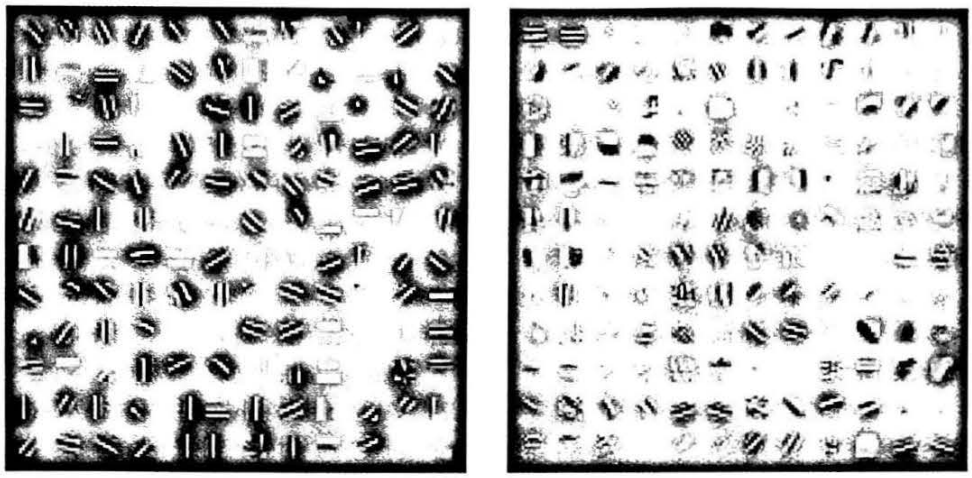


Figure 9.19: Many machine learning algorithms learn features that detect edges or specific colors of edges when applied to natural images. These feature detectors are reminiscent of the Gabor functions known to be present in primary visual cortex. *(Left)*Weights learned by an unsupervised learning algorithm (spike and slab sparse coding) applied to small image patches. *(Right)*Convolution kernels learned by the first layer of a fully supervised convolutional maxout network. Neighboring pairs of filters drive the same maxout unit.

networks to solve important commercial applications and remain at the forefront of commercial applications of deep learning today. For example, in the 1990s, the neural network research group at AT&T developed a convolutional network for reading checks (LeCun et al., 1998b). By the end of the 1990s, this system deployed by NEC was reading over 10 percent of all the checks in the United States. Later, several OCR and handwriting recognition systems based on convolutional nets were deployed by Microsoft (Simard et al., 2003). See chapter 12 for more details on such applications and more modern applications of convolutional networks. See LeCun et al. (2010) for a more in-depth history of convolutional networks up to 2010.

Convolutional networks were also used to win many contests. The current intensity of commercial interest in deep learning began when Krizhevsky et al. (2012) won the ImageNet object recognition challenge, but convolutional networks had been used to win other machine learning and computer vision contests with less impact for years earlier.

Convolutional nets were some of the first working deep networks trained with back-propagation. It is not entirely clear why convolutional networks succeeded when general back-propagation networks were considered to have failed. It may simply be that convolutional networks were more computationally efficient than fully connected networks, so it was easier to run multiple experiments with them and tune their implementation and hyperparameters. Larger networks also seem to

Microsoft Exhibit 1043 - Page 147
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

be easier to train. With modern hardware, large fully connected networks appear to perform reasonably on many tasks, even when using datasets that were available and activation functions that were popular during the times when fully connected networks were believed not to work well. It may be that the primary barriers to the success of neural networks were psychological (practitioners did not expect neural networks to work, so they did not make a serious effort to use neural networks). Whatever the case, it is fortunate that convolutional networks performed well decades ago. In many ways, they carried the torch for the rest of deep learning and paved the way to the acceptance of neural networks in general.

Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size. This approach has been the most successful on a two-dimensional image topology. To process one-dimensional sequential data, we turn next to another powerful specialization of the neural networks framework: recurrent neural networks.

Microsoft Exhibit 1043 - Page 148
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

10

Sequence Modeling: Recurrent and Recursive Nets

Recurrent neural networks, or RNNs (Rumelhart et al., 1986a), are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values $\mathbf{X}$ such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

To go from multilayer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters across different parts of a model. Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence. For example, consider the two sentences "I went to Nepal in 2009" and "In 2009, I went to Nepal." If we ask a machine learning model to read each sentence and

Microsoft Exhibit 1043 - Page 149
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

extract the year in which the narrator went to Nepal, we would like it to recognize the year 2009 as the relevant piece of information, whether it appears in the sixth word or in the second word of the sentence. Suppose that we trained a feedforward network that processes sentences of fixed length. A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all the rules of the language separately at each position in the sentence. By comparison, a recurrent neural network shares the same weights across several time steps.

A related idea is the use of convolution across a 1-D temporal sequence. This convolutional approach is the basis for time-delay neural networks (Lang and Hinton, 1988; Waibel et al., 1989; Lang et al., 1990). The convolution operation allows a network to share parameters across time but is shallow. The output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input. The idea of parameter sharing manifests in the application of the same convolution kernel at each time step. Recurrent networks share parameters in a different way. Each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.

For the simplicity of exposition, we refer to RNNs as operating on a sequence that contains vectors $\boldsymbol{x}^{(t)}$ with the time step index t ranging from 1 to τ. In practice, recurrent networks usually operate on minibatches of such sequences, with a different sequence length τ for each member of the minibatch. We have omitted the minibatch indices to simplify notation. Moreover, the time step index need not literally refer to the passage of time in the real world. Sometimes it refers only to the position in the sequence. RNNs may also be applied in two dimensions across spatial data such as images, and even when applied to data involving time, the network may have connections that go backward in time, provided that the entire sequence is observed before it is provided to the network.

This chapter extends the idea of a computational graph to include cycles. These cycles represent the influence of the present value of a variable on its own value at a future time step. Such computational graphs allow us to define recurrent neural networks. We then describe many different ways to construct, train, and use recurrent neural networks.

For more information on recurrent neural networks than is available in this chapter, we refer the reader to the textbook of Graves (2012).

Microsoft Exhibit 1043 - Page 150
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

10.1 Unfolding Computational Graphs

A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Please refer to section 6.5.1 for a general introduction. In this section we explain the idea of **unfolding** a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure.

For example, consider the classical form of a dynamical system:

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}), \tag{10.1}$$

where $\boldsymbol{s}^{(t)}$ is called the state of the system.

Equation 10.1 is recurrent because the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time $t-1$.

For a finite number of time steps τ, the graph can be unfolded by applying the definition $\tau - 1$ times. For example, if we unfold equation 10.1 for $\tau = 3$ time steps, we obtain

$$\begin{aligned} \boldsymbol{s}^{(3)} &= f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) && (10.2) \\ &= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}). && (10.3) \end{aligned}$$

Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph. The unfolded computational graph of equation 10.1 and equation 10.3 is illustrated in figure 10.1.

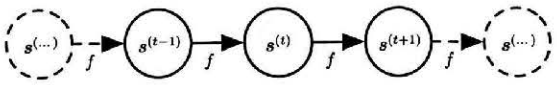


Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t, and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of $\boldsymbol{\theta}$ used to parametrize f) are used for all time steps.

Microsoft Exhibit 1043 - Page 151
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

As another example, let us consider a dynamical system driven by an external signal $\boldsymbol{x}^{(t)}$,

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \tag{10.4}$$

where we see that the state now contains information about the whole past sequence.

Recurrent neural networks can be built in many different ways. Much as almost any function can be considered a feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network.

Many recurrent neural networks use equation 10.5 or a similar equation to define the values of their hidden units. To indicate that the state is the hidden units of the network, we now rewrite equation 10.4 using the variable $\boldsymbol{h}$ to represent the state,

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \tag{10.5}$$

illustrated in figure 10.2; typical RNNs will add extra architectural features such as output layers that read information out of the state $\boldsymbol{h}$ to make predictions.

When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\boldsymbol{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t. This summary is in general necessarily lossy, since it maps an arbitrary length sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ to a fixed length vector $\boldsymbol{h}^{(t)}$. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects. For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, storing all the information in the input sequence up to time t may not be necessary; storing only enough information to predict the rest of the sentence is sufficient. The most demanding situation is when we ask $\boldsymbol{h}^{(t)}$ to be rich

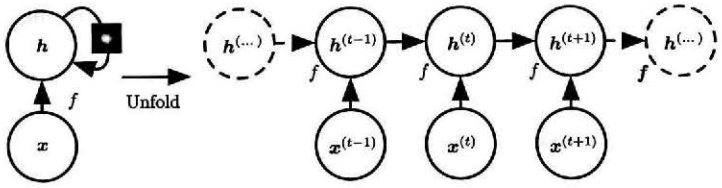


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of a single time step. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

Microsoft Exhibit 1043 - Page 152
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

enough to allow one to approximately recover the input sequence, as in autoencoder frameworks (chapter 14).

Equation 10.5 can be drawn in two different ways. One way to draw the RNN is with a diagram containing one node for every component that might exist in a physical implementation of the model, such as a biological neural network. In this view, the network defines a circuit that operates in real time, with physical parts whose current state can influence their future state, as in the left of figure 10.2. Throughout this chapter, we use a black square in a circuit diagram to indicate that an interaction takes place with a delay of a single time step, from the state at time t to the state at time $t+1$. The other way to draw the RNN is as an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that point in time. Each variable for each time step is drawn as a separate node of the computational graph, as in the right of figure 10.2. What we call unfolding is the operation that maps a circuit, as in the left side of the figure, to a computational graph with repeated pieces, as in the right side. The unfolded graph now has a size that depends on the sequence length.

We can represent the unfolded recurrence after t steps with a function $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \tag{10.6}$$

$$= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}). \tag{10.7}$$

The function $g^{(t)}$ takes the whole past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f. The unfolding process thus introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.

2. It is possible to use the *same* transition function f with the same parameters at every time step.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model $g^{(t)}$ for all possible time steps. Learning a single shared model allows generalization to sequence lengths that did not appear in the training set, and enables the model

Microsoft Exhibit 1043 - Page 153
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

to be estimated with far fewer training examples than would be required without parameter sharing.

Both the recurrent graph and the unrolled graph have their uses. The recurrent graph is succinct. The unfolded graph provides an explicit description of which computations to perform. The unfolded graph also helps illustrate the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.

10.2 Recurrent Neural Networks

Armed with the graph-unrolling and parameter-sharing ideas of section 10.1, we can design a wide variety of recurrent neural networks.

Some examples of important design patterns for recurrent neural networks include the following:

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in figure 10.3
- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step, illustrated in figure 10.4
- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output, illustrated in figure 10.5

Figure 10.3 is a reasonably representative example that we return to throughout most of the chapter.

The recurrent neural network of figure 10.3 and equation 10.8 is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size. The output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input (Siegelmann and Sontag, 1991; Siegelmann, 1995; Siegelmann and Sontag, 1995; Hyotyniemi, 1996). The functions computable by a Turing machine are discrete, so these results regard exact implementation of the function, not approximations. The RNN, when used as a Turing machine, takes a binary sequence as input, and its outputs must be discretized to provide a binary output. It is possible to

Microsoft Exhibit 1043 - Page 154
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

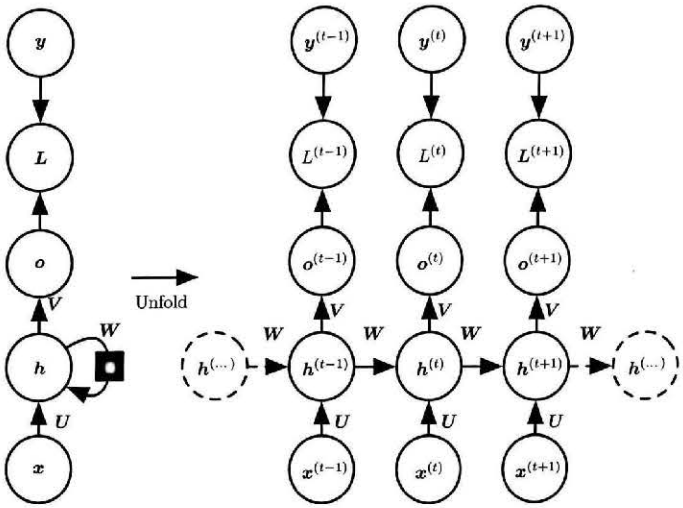


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Equation 10.8 defines forward propagation in this model. *(Left)* The RNN and its loss drawn with recurrent connections. *(Right)* The same seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.

compute all functions in this setting using a single specific RNN of finite size (Siegelmann and Sontag[1995] use 886 units). The "input" of the Turing machine is a specification of the function to be computed, so the same network that simulates this Turing machine is sufficient for all problems. The theoretical RNN used for the proof can simulate an unbounded stack by representing its activations and weights with rational numbers of unbounded precision.

We now develop the forward propagation equations for the RNN depicted in figure 10.3. The figure does not specify the choice of activation function for the hidden units. Here we assume the hyperbolic tangent activation function. Also, the figure does not specify exactly what form the output and loss function take. Here we assume that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent discrete variables is to regard the output $\boldsymbol{o}$

Microsoft Exhibit 1043 - Page 155
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

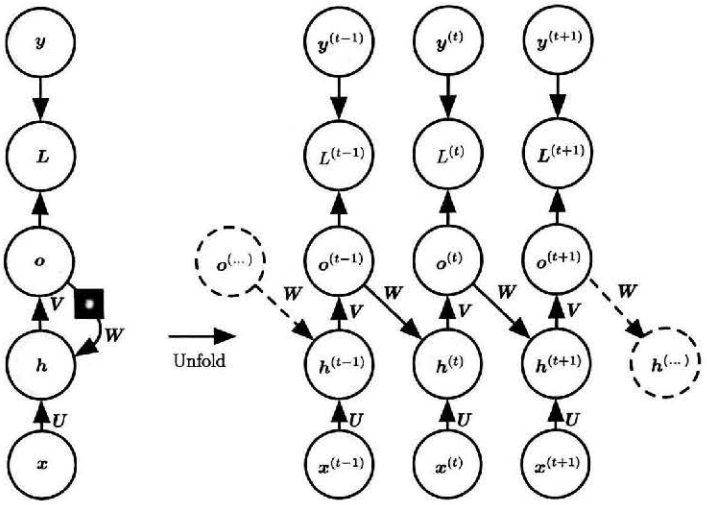


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$, and the loss is $L^{(t)}$. *(Left)* Circuit diagram. *(Right)* Unfolded computational graph. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by figure 10.3. The RNN in figure 10.3 can choose to put any information it wants about the past into its hidden representation $\boldsymbol{h}$ and transmit $\boldsymbol{h}$ to the future. The RNN in this figure is trained to put a specific output value into $\boldsymbol{o}$, and $\boldsymbol{o}$ is the only information it is allowed to send to the future. There are no direct connections from $\boldsymbol{h}$ going forward. The previous $\boldsymbol{h}$ is connected to the present only indirectly, via the predictions it was used to produce. Unless $\boldsymbol{o}$ is very high-dimensional and rich, it will usually lack important information from the past. This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training, as described in section 10.2.1.

as giving the unnormalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector $\hat{\boldsymbol{y}}$ of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \tag{10.8}$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}), \tag{10.9}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \tag{10.10}$$

Microsoft Exhibit 1043 - Page 156
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

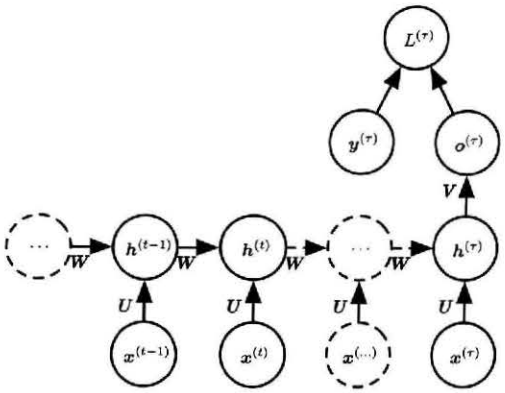


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here), or the gradient on the output $\boldsymbol{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}), \tag{10.11}$$

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of $\boldsymbol{x}$ values paired with a sequence of $\boldsymbol{y}$ values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$, then

$$L\left(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\right) \tag{10.12}$$

$$= \sum_t L^{(t)} \tag{10.13}$$

$$= -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right), \tag{10.14}$$

where $p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right)$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\boldsymbol{y}}^{(t)}$. Computing the gradient of this loss function with respect to the parameters is an expensive operation. The gradient computation involves performing a forward propagation pass moving left to right through our illustration

Microsoft Exhibit 1043 - Page 157
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

of the unrolled graph in figure 10.3, followed by a backward propagation pass moving right to left through the graph. The runtime is $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$. The back-propagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called **back-propagation through time** (BPTT) and is discussed further in section 10.2.2. The network with recurrence between hidden units is thus very powerful but also expensive to train. Is there an alternative?

10.2.1 Teacher Forcing and Networks with Output Recurrence

The network with recurrent connections only from the output at one time step to the hidden units at the next time step (shown in figure 10.4) is strictly less powerful because it lacks hidden-to-hidden recurrent connections. For example, it cannot simulate a universal Turing machine. Because this network lacks hidden-to-hidden recurrence, it requires that the output units capture all the information about the past that the network will use to predict the future. Because the output units are explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets. The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled. Training can thus be parallelized, with the gradient for each step t computed in isolation. There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

Models that have recurrent connections from their outputs leading back into the model may be trained with **teacher forcing**. Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t+1$. We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is

$$\log p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) \tag{10.15}$$

$$= \log p\left(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) + \log p\left(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right). \tag{10.16}$$

Microsoft Exhibit 1043 - Page 158
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

13

Linear Factor Models

Many of the research frontiers in deep learning involve building a probabilistic model of the input, $p_{\text{model}}(\boldsymbol{x})$. Such a model can, in principle, use probabilistic inference to predict any of the variables in its environment given any of the other variables. Many of these models also have latent variables $\boldsymbol{h}$, with $p_{\text{model}}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{h}} p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{h})$. These latent variables provide another means of representing the data. Distributed representations based on latent variables can obtain all the advantages of representation learning that we have seen with deep feedforward and recurrent networks.

In this chapter, we describe some of the simplest probabilistic models with latent variables: linear factor models. These models are sometimes used as building blocks of mixture models (Hinton et al., 1995a; Ghahramani and Hinton, 1996; Roweis et al., 2002) or of larger, deep probabilistic models (Tang et al., 2012). They also show many of the basic approaches necessary to building generative models that the more advanced deep models will extend further.

A linear factor model is defined by the use of a stochastic linear decoder function that generates $\boldsymbol{x}$ by adding noise to a linear transformation of $\boldsymbol{h}$.

These models are interesting because they allow us to discover explanatory factors that have a simple joint distribution. The simplicity of using a linear decoder made these models some of the first latent variable models to be extensively studied.

A linear factor model describes the data-generation process as follows. First, we sample the explanatory factors $\boldsymbol{h}$ from a distribution

$$\mathbf{h} \sim p(\boldsymbol{h}), \tag{13.1}$$

Microsoft Exhibit 1043 - Page 159
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

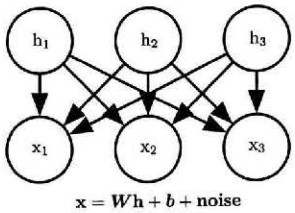


Figure 13.1: The directed graphical model describing the linear factor model family, in which we assume that an observed data vector $\boldsymbol{x}$ is obtained by a linear combination of independent latent factors $\boldsymbol{h}$, plus some noise. Different models, such as probabilistic PCA, factor analysis or ICA, make different choices about the form of the noise and of the prior $p(\boldsymbol{h})$.

where $p(\boldsymbol{h})$ is a factorial distribution, with $p(\boldsymbol{h}) = \prod_i p(h_i)$, so that it is easy to sample from. Next we sample the real-valued observable variables given the factors

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise}, \tag{13.2}$$

where the noise is typically Gaussian and diagonal (independent across dimensions). This is illustrated in figure 13.1.

13.1 Probabilistic PCA and Factor Analysis

Probabilistic PCA (principal components analysis), factor analysis and other linear factor models are special cases of the above equations (13.1 and 13.2) and only differ in the choices made for the noise distribution and the model's prior over latent variables $\boldsymbol{h}$ before observing $\boldsymbol{x}$.

In **factor analysis** (Bartholomew, 1987; Basilevsky, 1994), the latent variable prior is just the unit variance Gaussian

$$\mathbf{h} \sim \mathcal{N}(\boldsymbol{h}; \mathbf{0}, \boldsymbol{I}), \tag{13.3}$$

while the observed variables x_i are assumed to be **conditionally independent**, given $\boldsymbol{h}$. Specifically, the noise is assumed to be drawn from a diagonal covariance Gaussian distribution, with covariance matrix $\boldsymbol{\psi} = \text{diag}(\boldsymbol{\sigma}^2)$, with $\boldsymbol{\sigma}^2 = [\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2]^\top$ a vector of per-variable variances.

The role of the latent variables is thus to *capture the dependencies* between the different observed variables x_i. Indeed, it can easily be shown that $\boldsymbol{x}$ is just a multivariate normal random variable, with

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \boldsymbol{\psi}). \tag{13.4}$$

Microsoft Exhibit 1043 - Page 160
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

To cast PCA in a probabilistic framework, we can make a slight modification to the factor analysis model, making the conditional variances σ_i^2 equal to each other. In that case the covariance of $\boldsymbol{x}$ is just $\boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}$, where σ^2 is now a scalar. This yields the conditional distribution

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}), \tag{13.5}$$

or equivalently

$$\mathbf{x} = \boldsymbol{W}\mathbf{h} + \boldsymbol{b} + \sigma\mathbf{z}, \tag{13.6}$$

where $\mathbf{z} \sim \mathcal{N}(\boldsymbol{z}; \mathbf{0}, \boldsymbol{I})$ is Gaussian noise. Then, as Tipping and Bishop (1999) show, we can use an iterative EM algorithm for estimating the parameters $\boldsymbol{W}$ and σ^2.

This **probabilistic** PCA model takes advantage of the observation that most variations in the data can be captured by the latent variables $\boldsymbol{h}$, up to some small residual **reconstruction error** σ^2. As shown by Tipping and Bishop (1999), probabilistic PCA becomes PCA as $\sigma \to 0$. In that case, the conditional expected value of $\boldsymbol{h}$ given $\boldsymbol{x}$ becomes an orthogonal projection of $\boldsymbol{x} - \boldsymbol{b}$ onto the space spanned by the d columns of $\boldsymbol{W}$, as in PCA.

As $\sigma \to 0$, the density model defined by probabilistic PCA becomes very sharp around these d dimensions spanned by the columns of $\boldsymbol{W}$. This can make the model assign very low likelihood to the data if the data do not actually cluster near a hyperplane.

13.2 Independent Component Analysis (ICA)

Independent component analysis (ICA) is among the oldest representation learning algorithms (Herault and Ans, 1984; Jutten and Herault, 1991; Comon, 1994; Hyvärinen, 1999; Hyvärinen et al., 2001a; Hinton et al., 2001; Teh et al., 2003). It is an approach to modeling linear factors that seeks to separate an observed signal into many underlying signals that are scaled and added together to form the observed data. These signals are intended to be fully independent, rather than merely decorrelated from each other.[1]

Many different specific methodologies are referred to as ICA. The variant that is most similar to the other generative models we have described here is a variant (Pham et al., 1992) that trains a fully parametric generative model. The prior distribution over the underlying factors, $p(\boldsymbol{h})$, must be fixed ahead of time by

[1] See section 3.8 for a discussion of the difference between uncorrelated variables and independent variables.

Microsoft Exhibit 1043 - Page 161
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

the user. The model then deterministically generates $\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h}$. We can perform a nonlinear change of variables (using equation 3.47) to determine $p(\boldsymbol{x})$. Learning the model then proceeds as usual, using maximum likelihood.

The motivation for this approach is that by choosing $p(\boldsymbol{h})$ to be independent, we can recover underlying factors that are as close as possible to independent. This is commonly used, not to capture high-level abstract causal factors, but to recover low-level signals that have been mixed together. In this setting, each training example is one moment in time, each x_i is one sensor's observation of the mixed signals, and each h_i is one estimate of one of the original signals. For example, we might have n people speaking simultaneously. If we have n different microphones placed in different locations, ICA can detect the changes in the volume between each speaker as heard by each microphone and separate the signals so that each h_i contains only one person speaking clearly. This is commonly used in neuroscience for electroencephalography, a technology for recording electrical signals originating in the brain. Multiple electrode sensors placed on the subject's head are used to measure many electrical signals coming from the body. The experimenter is typically only interested in signals from the brain, but signals from the subject's heart and eyes are strong enough to confound measurements taken at the subject's scalp. The signals arrive at the electrodes mixed together, so ICA is necessary to separate the electrical signature of the heart from the signals originating in the brain, and to separate signals in different brain regions from each other.

As mentioned before, many variants of ICA are possible. Some add some noise in the generation of $\boldsymbol{x}$ rather than using a deterministic decoder. Most do not use the maximum likelihood criterion, but instead aim to make the elements of $\boldsymbol{h} = \boldsymbol{W}^{-1}\boldsymbol{x}$ independent from each other. Many criteria that accomplish this goal are possible. Equation 3.47 requires taking the determinant of $\boldsymbol{W}$, which can be an expensive and numerically unstable operation. Some variants of ICA avoid this problematic operation by constraining $\boldsymbol{W}$ to be orthogonal.

All variants of ICA require that $p(\boldsymbol{h})$ be non-Gaussian. This is because if $p(\boldsymbol{h})$ is an independent prior with Gaussian components, then $\boldsymbol{W}$ is not identifiable. We can obtain the same distribution over $p(\boldsymbol{x})$ for many values of $\boldsymbol{W}$. This is very different from other linear factor models like probabilistic PCA and factor analysis, which often require $p(\boldsymbol{h})$ to be Gaussian in order to make many operations on the model have closed form solutions. In the maximum likelihood approach, where the user explicitly specifies the distribution, a typical choice is to use $p(h_i) = \frac{d}{dh_i}\sigma(h_i)$. Typical choices of these non-Gaussian distributions have larger peaks near 0 than

Microsoft Exhibit 1043 - Page 162
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

does the Gaussian distribution, so we can also see most implementations of ICA as learning sparse features.

Many variants of ICA are not generative models in the sense that we use the phrase. In this book, a generative model either represents $p(\boldsymbol{x})$ or can draw samples from it. Many variants of ICA only know how to transform between $\boldsymbol{x}$ and $\boldsymbol{h}$ but do not have any way of representing $p(\boldsymbol{h})$, and thus do not impose a distribution over $p(\boldsymbol{x})$. For example, many ICA variants aim to increase the sample kurtosis of $\boldsymbol{h} = \boldsymbol{W}^{-1}\boldsymbol{x}$, because high kurtosis indicates that $p(\boldsymbol{h})$ is non-Gaussian, but this is accomplished without explicitly representing $p(\boldsymbol{h})$. This is because ICA is more often used as an analysis tool for separating signals, rather than for generating data or estimating its density.

Just as PCA can be generalized to the nonlinear autoencoders described in chapter 14, ICA can be generalized to a nonlinear generative model, in which we use a nonlinear function f to generate the observed data. See Hyvärinen and Pajunen (1999) for the initial work on nonlinear ICA and its successful use with ensemble learning by Roberts and Everson (2001) and Lappalainen et al. (2000). Another nonlinear extension of ICA is the approach of **nonlinear independent components estimation**, or NICE (Dinh et al., 2014), which stacks a series of invertible transformations (encoder stages) with the property that the determinant of the Jacobian of each transformation can be computed efficiently. This makes it possible to compute the likelihood exactly, and like ICA, NICE attempts to transform the data into a space where it has a factorized marginal distribution, but it is more likely to succeed thanks to the nonlinear encoder. Because the encoder is associated with a decoder that is its perfect inverse, generating samples from the model is straightforward (by first sampling from $p(\boldsymbol{h})$ and then applying the decoder).

Another generalization of ICA is to learn groups of features, with statistical dependence allowed within a group but discouraged between groups (Hyvärinen and Hoyer, 1999; Hyvärinen et al., 2001b). When the groups of related units are chosen to be nonoverlapping, this is called **independent subspace analysis**. It is also possible to assign spatial coordinates to each hidden unit and form overlapping groups of spatially neighboring units. This encourages nearby units to learn similar features. When applied to natural images, this **topographic ICA** approach learns Gabor filters, such that neighboring features have similar orientation, location or frequency. Many different phase offsets of similar Gabor functions occur within each region, so that pooling over small regions yields translation invariance.

Microsoft Exhibit 1043 - Page 163
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

13.3 Slow Feature Analysis

Slow feature analysis (SFA) is a linear factor model that uses information from time signals to learn invariant features (Wiskott and Sejnowski, 2002).

Slow feature analysis is motivated by a general principle called the slowness principle. The idea is that the important characteristics of scenes change very slowly compared to the individual measurements that make up a description of a scene. For example, in computer vision, individual pixel values can change very rapidly. If a zebra moves from left to right across the image, an individual pixel will rapidly change from black to white and back again as the zebra's stripes pass over the pixel. By comparison, the feature indicating whether a zebra is in the image will not change at all, and the feature describing the zebra's position will change slowly. We therefore may wish to regularize our model to learn features that change slowly over time.

The slowness principle predates slow feature analysis and has been applied to a wide variety of models (Hinton, 1989; Földiák, 1989; Mobahi et al., 2009; Bergstra and Bengio, 2009). In general, we can apply the slowness principle to any differentiable model trained with gradient descent. The slowness principle may be introduced by adding a term to the cost function of the form

$$\lambda \sum_t L(f(\boldsymbol{x}^{(t+1)}), f(\boldsymbol{x}^{(t)})), \tag{13.7}$$

where λ is a hyperparameter determining the strength of the slowness regularization term, t is the index into a time sequence of examples, f is the feature extractor to be regularized, and L is a loss function measuring the distance between $f(\boldsymbol{x}^{(t)})$ and $f(\boldsymbol{x}^{(t+1)})$. A common choice for L is the mean squared difference.

Slow feature analysis is a particularly efficient application of the slowness principle. It is efficient because it is applied to a linear feature extractor and can thus be trained in closed form. Like some variants of ICA, SFA is not quite a generative model per se, in the sense that it defines a linear map between input space and feature space but does not define a prior over feature space and thus does not impose a distribution $p(\boldsymbol{x})$ on input space.

The SFA algorithm (Wiskott and Sejnowski, 2002) consists of defining $f(\boldsymbol{x}; \boldsymbol{\theta})$ to be a linear transformation, then solving the optimization problem

$$\min_{\boldsymbol{\theta}} \mathbb{E}_t (f(\boldsymbol{x}^{(t+1)})_i - f(\boldsymbol{x}^{(t)})_i)^2 \tag{13.8}$$

subject to the constraints

$$\mathbb{E}_t f(\boldsymbol{x}^{(t)})_i = 0 \tag{13.9}$$

Microsoft Exhibit 1043 - Page 164
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

14

Autoencoders

An **autoencoder** is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer $\boldsymbol{h}$ that describes a **code** used to represent the input. The network may be viewed as consisting of two parts: an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder that produces a reconstruction $\boldsymbol{r} = g(\boldsymbol{h})$. This architecture is presented in figure 14.1. If an autoencoder succeeds in simply learning to set $g(f(\boldsymbol{x})) = \boldsymbol{x}$ everywhere, then it is not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ and $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$.

The idea of autoencoders has been part of the historical landscape of neural networks for decades (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994). Traditionally, autoencoders were used for dimensionality reduction or feature learning. Recently, theoretical connections between autoencoders and latent variable models have brought autoencoders to the forefront of generative modeling, as we will see in chapter 20. Autoencoders may be thought of as being a special case of feedforward networks and may be trained with all the same techniques, typically minibatch gradient descent following gradients computed by back-propagation. Unlike general feedforward networks, autoencoders may also be trained using **recirculation** (Hinton and McClelland, 1988), a learning algorithm based on comparing the activations of the network on the original input to the activations

Microsoft Exhibit 1043 - Page 165
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

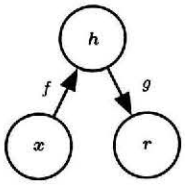


Figure 14.1: The general structure of an autoencoder, mapping an input $\boldsymbol{x}$ to an output (called reconstruction) $\boldsymbol{r}$ through an internal representation or code $\boldsymbol{h}$. The autoencoder has two components: the encoder f (mapping $\boldsymbol{x}$ to $\boldsymbol{h}$) and the decoder g (mapping $\boldsymbol{h}$ to $\boldsymbol{r}$).

on the reconstructed input. Recirculation is regarded as more biologically plausible than back-propagation but is rarely used for machine learning applications.

14.1 Undercomplete Autoencoders

Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in $\boldsymbol{h}$ taking on useful properties.

One way to obtain useful features from the autoencoder is to constrain $\boldsymbol{h}$ to have a smaller dimension than $\boldsymbol{x}$. An autoencoder whose code dimension is less than the input dimension is called **undercomplete**. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))), \tag{14.1}$$

where L is a loss function penalizing $g(f(\boldsymbol{x}))$ for being dissimilar from $\boldsymbol{x}$, such as the mean squared error.

When the decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA. In this case, an autoencoder trained to perform the copying task has learned the principal subspace of the training data as a side effect.

Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA. Unfortunately, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information about

Microsoft Exhibit 1043 - Page 166
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

the distribution of the data. Theoretically, one could imagine that an autoencoder with a one-dimensional code but a very powerful nonlinear encoder could learn to represent each training example $\boldsymbol{x}^{(i)}$ with the code i. The decoder could learn to map these integer indices back to the values of specific training examples. This specific scenario does not occur in practice, but it illustrates clearly that an autoencoder trained to perform the copying task can fail to learn anything useful about the dataset if the capacity of the autoencoder is allowed to become too great.

14.2 Regularized Autoencoders

Undercomplete autoencoders, with code dimension less than the input dimension, can learn the most salient features of the data distribution. We have seen that these autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity.

A similar problem occurs if the hidden code is allowed to have dimension equal to the input, and in the **overcomplete** case in which the hidden code has dimension greater than the input. In these cases, even a linear encoder and a linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.

Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled. Regularized autoencoders provide the ability to do so. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output. These other properties include sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs. A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn a trivial identity function.

In addition to the methods described here, which are most naturally interpreted as regularized autoencoders, nearly any generative model with latent variables and equipped with an inference procedure (for computing latent representations given input) may be viewed as a particular form of autoencoder. Two generative modeling approaches that emphasize this connection with autoencoders are the descendants of the Helmholtz machine (Hinton et al., 1995b), such as the variational autoencoder (section 20.10.3) and the generative stochastic networks (section 20.12). These

Microsoft Exhibit 1043 - Page 167
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

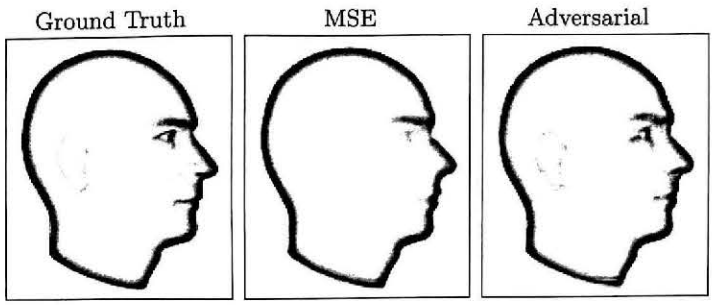


Figure 15.6: Predictive generative networks provide an example of the importance of learning which features are salient. In this example, the predictive generative network has been trained to predict the appearance of a 3-D model of a human head at a specific viewing angle. *(Left)*Ground truth. This is the correct image, which the network should emit. *(Center)*Image produced by a predictive generative network trained with mean squared error alone. Because the ears do not cause an extreme difference in brightness compared to the neighboring skin, they were not sufficiently salient for the model to learn to represent them. *(Right)*Image produced by a model trained with a combination of mean squared error and adversarial loss. Using this learned cost function, the ears are salient because they follow a predictable pattern. Learning which underlying causes are important and relevant enough to model is an important active area of research. Figures graciously provided by Lotter et al. (2015).

distribution due to different domains, temporal nonstationarity, or changes in the nature of the task, *the causal mechanisms remain invariant* ("the laws of the universe are constant"), while the marginal distribution over the underlying causes can change. Hence, better generalization and robustness to all kinds of changes can be expected via learning a generative model that attempts to recover the causal factors $\mathbf{h}$ and $p(\mathbf{x} \mid \mathbf{h})$.

15.4 Distributed Representation

Distributed representations of concepts—representations composed of many elements that can be set separately from each other—are one of the most important tools for representation learning. Distributed representations are powerful because they can use n features with k values to describe k^n different concepts. As we have seen throughout this book, neural networks with multiple hidden units and probabilistic models with multiple latent variables both make use of the strategy of distributed representation. We now introduce an additional observation. Many deep

Microsoft Exhibit 1043 - Page 168
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

learning algorithms are motivated by the assumption that the hidden units can learn to represent the underlying causal factors that explain the data, as discussed in section 15.3. Distributed representations are natural for this approach, because each direction in representation space can correspond to the value of a different underlying configuration variable.

An example of a distributed representation is a vector of n binary features, which can take 2^n configurations, each potentially corresponding to a different region in input space, as illustrated in figure 15.7. This can be compared with a *symbolic representation*, where the input is associated with a single symbol or category. If there are n symbols in the dictionary, one can imagine n feature detectors, each corresponding to the detection of the presence of the associated category. In that case only n different configurations of the representation space are possible, carving n different regions in input space, as illustrated in figure 15.8. Such a symbolic representation is also called a one-hot representation, since it can be captured by a binary vector with n bits that are mutually exclusive (only one of them can be active). A symbolic representation is a specific example of the broader class of nondistributed representations, which are representations that may contain many entries but without significant meaningful separate control over each entry.

The following examples of learning algorithms are based on nondistributed representations:

- Clustering methods, including the k-means algorithm: each input point is assigned to exactly one cluster.
- k-nearest neighbors algorithms: one or a few templates or prototype examples are associated with a given input. In the case of $k > 1$, multiple values describe each input, but they cannot be controlled separately from each other, so this does not qualify as a true distributed representation.
- Decision trees: only one leaf (and the nodes on the path from root to leaf) is activated when an input is given.
- Gaussian mixtures and mixtures of experts: the templates (cluster centers) or experts are now associated with a *degree* of activation. As with the k-nearest neighbors algorithm, each input is represented with multiple values, but those values cannot readily be controlled separately from each other.
- Kernel machines with a Gaussian kernel (or other similarly local kernel): although the degree of activation of each "support vector" or template example is now continuous-valued, the same issue arises as with Gaussian mixtures.

Microsoft Exhibit 1043 - Page 169
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

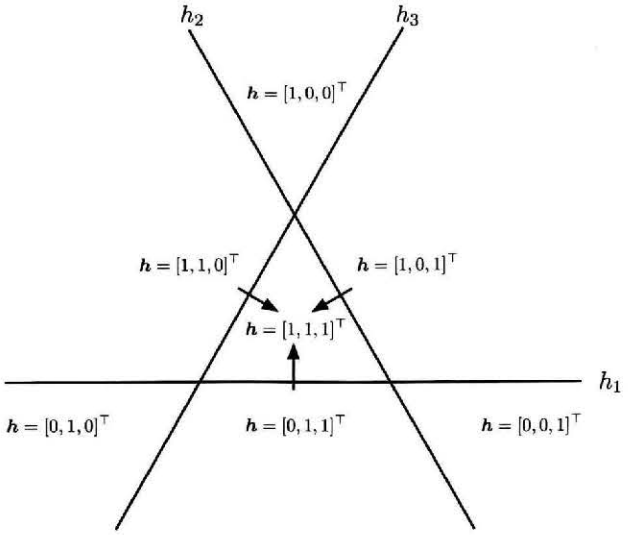


Figure 15.7: Illustration of how a learning algorithm based on a distributed representation breaks up the input space into regions. In this example, there are three binary features h_1, h_2, and h_3. Each feature is defined by thresholding the output of a learned linear transformation. Each feature divides $\mathbb{R}^2$ into two half-planes. Let h_i^+ be the set of input points for which $h_i = 1$, and h_i^- be the set of input points for which $h_i = 0$. In this illustration, each line represents the decision boundary for one h_i, with the corresponding arrow pointing to the h_i^+ side of the boundary. The representation as a whole takes on a unique value at each possible intersection of these half-planes. For example, the representation value $[1, 1, 1]^\top$ corresponds to the region $h_1^+ \cap h_2^+ \cap h_3^+$. Compare this to the non-distributed representations in figure 15.8. In the general case of d input dimensions, a distributed representation divides $\mathbb{R}^d$ by intersecting half-spaces rather than half-planes. The distributed representation with n features assigns unique codes to $O(n^d)$ different regions, while the nearest neighbor algorithm with n examples assigns unique codes to only n regions. The distributed representation is thus able to distinguish exponentially many more regions than the nondistributed one. Keep in mind that not all $\boldsymbol{h}$ values are feasible (there is no $\boldsymbol{h} = \mathbf{0}$ in this example), and that a linear classifier on top of the distributed representation is not able to assign different class identities to every neighboring region; even a deep linear-threshold network has a VC dimension of only $O(w \log w)$, where w is the number of weights (Sontag, 1998). The combination of a powerful representation layer and a weak classifier layer can be a strong regularizer; a classifier trying to learn the concept of "person" versus "not a person" does not need to assign a different class to an input represented as "woman with glasses" than it assigns to an input represented as "man without glasses." This capacity constraint encourages each classifier to focus on few h_i and encourages $\boldsymbol{h}$ to learn to represent the classes in a linearly separable way.

Microsoft Exhibit 1043 - Page 170
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

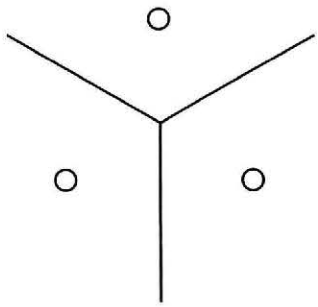


Figure 15.8: Illustration of how the nearest neighbor algorithm breaks up the input space into different regions. The nearest neighbor algorithm provides an example of a learning algorithm based on a nondistributed representation. Different non-distributed algorithms may have different geometry, but they typically break the input space into regions, *with a separate set of parameters for each region.* The advantage of a nondistributed approach is that, given enough parameters, it can fit the training set without solving a difficult optimization algorithm, because it is straightforward to choose a different output *independently* for each region. The disadvantage is that such nondistributed models generalize only locally via the smoothness prior, making it difficult to learn a complicated function with more peaks and troughs than the available number of examples. Contrast this with a distributed representation, figure 15.7.

- Language or translation models based on n-grams: the set of contexts (sequences of symbols) is partitioned according to a tree structure of suffixes. A leaf may correspond to the last two words being w_1 and w_2, for example. Separate parameters are estimated for each leaf of the tree (with some sharing being possible).

For some of these nondistributed algorithms, the output is not constant by parts but instead interpolates between neighboring regions. The relationship between the number of parameters (or examples) and the number of regions they can define remains linear.

An important related concept that distinguishes a distributed representation from a symbolic one is that *generalization arises due to shared attributes* between different concepts. As pure symbols, "`cat`" and "`dog`" are as far from each other as any other two symbols. However, if one associates them with a meaningful distributed representation, then many of the things that can be said about cats can generalize to dogs and vice versa. For example, our distributed representation may contain entries such as "`has_fur`" or "`number_of_legs`" that have the same value for the embedding of both "`cat`" and "`dog`." Neural language models that

Microsoft Exhibit 1043 - Page 171
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

operate on distributed representations of words generalize much better than other models that operate directly on one-hot representations of words, as discussed in section 12.4. Distributed representations induce a rich *similarity space*, in which semantically close concepts (or inputs) are close in distance, a property that is absent from purely symbolic representations.

When and why can there be a statistical advantage from using a distributed representation as part of a learning algorithm? Distributed representations can have a statistical advantage when an apparently complicated structure can be compactly represented using a small number of parameters. Some traditional nondistributed learning algorithms generalize only due to the smoothness assumption, which states that if $u \approx v$, then the target function f to be learned has the property that $f(u) \approx f(v)$ in general. There are many ways of formalizing such an assumption, but the end result is that if we have an example (x, y) for which we know that $f(x) \approx y$, then we choose an estimator $\hat{f}$ that approximately satisfies these constraints while changing as little as possible when we move to a nearby input $x + \epsilon$. This assumption is clearly very useful, but it suffers from the curse of dimensionality: to learn a target function that increases and decreases many times in many different regions,[1] we may need a number of examples that is at least as large as the number of distinguishable regions. One can think of each of these regions as a category or symbol: by having a separate degree of freedom for each symbol (or region), we can learn an arbitrary decoder mapping from symbol to value. However, this does not allow us to generalize to new symbols for new regions.

If we are lucky, there may be some regularity in the target function, besides being smooth. For example, a convolutional network with max pooling can recognize an object regardless of its location in the image, even though spatial translation of the object may not correspond to smooth transformations in the input space.

Let us examine a special case of a distributed representation learning algorithm, which extracts binary features by thresholding linear functions of the input. Each binary feature in this representation divides $\mathbb{R}^d$ into a pair of half-spaces, as illustrated in figure 15.7. The exponentially large number of intersections of n of the corresponding half-spaces determines how many regions this distributed representation learner can distinguish. How many regions are generated by an arrangement of n hyperplanes in $\mathbb{R}^d$? By applying a general result concerning the

[1] Potentially, we may want to learn a function whose behavior is distinct in exponentially many regions: in a d-dimensional space with at least 2 different values to distinguish per dimension, we might want f to differ in 2^d different regions, requiring $O(2^d)$ training examples.

Microsoft Exhibit 1043 - Page 172
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

intersection of hyperplanes (Zaslavsky, 1975), one can show (Pascanu et al., 2014b) that the number of regions this binary feature representation can distinguish is

$$\sum_{j=0}^{d} \binom{n}{j} = O(n^d). \tag{15.4}$$

Therefore, we see a growth that is exponential in the input size and polynomial in the number of hidden units.

This provides a geometric argument to explain the generalization power of distributed representation: with $O(nd)$ parameters (for n linear threshold features in $\mathbb{R}^d$), we can distinctly represent $O(n^d)$ regions in input space. If instead we made no assumption at all about the data, and used a representation with one unique symbol for each region, and separate parameters for each symbol to recognize its corresponding portion of $\mathbb{R}^d$, then specifying $O(n^d)$ regions would require $O(n^d)$ examples. More generally, the argument in favor of the distributed representation could be extended to the case where instead of using linear threshold units we use nonlinear, possibly continuous, feature extractors for each of the attributes in the distributed representation. The argument in this case is that if a parametric transformation with k parameters can learn about r regions in input space, with $k \ll r$, and if obtaining such a representation was useful to the task of interest, then we could potentially generalize much better in this way than in a nondistributed setting, where we would need $O(r)$ examples to obtain the same features and associated partitioning of the input space into r regions. Using fewer parameters to represent the model means that we have fewer parameters to fit, and thus require far fewer training examples to generalize well.

A further part of the argument for why models based on distributed representations generalize well is that their capacity remains limited despite being able to distinctly encode so many different regions. For example, the VC dimension of a neural network of linear threshold units is only $O(w \log w)$, where w is the number of weights (Sontag, 1998). This limitation arises because, while we can assign very many unique codes to representation space, we cannot use absolutely all the code space, nor can we learn arbitrary functions mapping from the representation space $\boldsymbol{h}$ to the output $\boldsymbol{y}$ using a linear classifier. The use of a distributed representation combined with a linear classifier thus expresses a prior belief that the classes to be recognized are linearly separable as a function of the underlying causal factors captured by $\boldsymbol{h}$. We will typically want to learn categories such as the set of all images of all green objects or the set of all images of cars, but not categories that require nonlinear XOR logic. For example, we typically do not want to partition

Microsoft Exhibit 1043 - Page 173
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

the data into the set of all red cars and green trucks as one class and the set of all green cars and red trucks as another class.

The ideas discussed so far have been abstract, but they may be experimentally validated. Zhou et al. (2015) find that hidden units in a deep convolutional network trained on the ImageNet and Places benchmark datasets learn features that are often interpretable, corresponding to a label that humans would naturally assign. In practice it is certainly not always the case that hidden units learn something that has a simple linguistic name, but it is interesting to see this emerge near the top levels of the best computer vision deep networks. What such features have in common is that one could imagine *learning about each of them without having to see all the configurations of all the others.* Radford et al. (2015) demonstrated that a generative model can learn a representation of images of faces, with separate directions in representation space capturing different underlying factors of variation. Figure 15.9 demonstrates that one direction in representation space corresponds to whether the person is male or female, while another corresponds to whether the person is wearing glasses. These features were discovered automatically, not fixed a priori. There is no need to have labels for the hidden unit classifiers: gradient descent on an objective function of interest naturally learns semantically interesting features, as long as the task requires such features. We can learn about

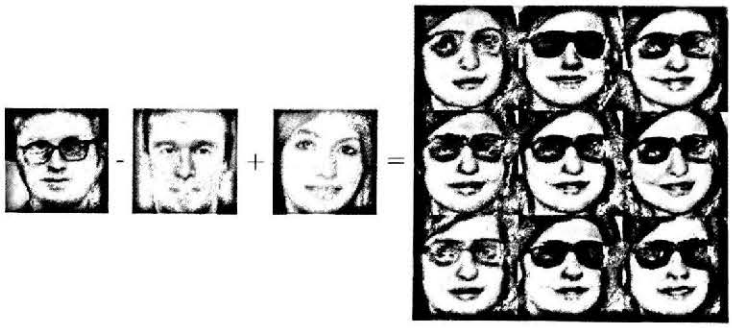


Figure 15.9: A generative model has learned a distributed representation that disentangles the concept of gender from the concept of wearing glasses. If we begin with the representation of the concept of a man with glasses, then subtract the vector representing the concept of a man without glasses, and finally add the vector representing the concept of a woman without glasses, we obtain the vector representing the concept of a woman with glasses. The generative model correctly decodes all these representation vectors to images that may be recognized as belonging to the correct class. Images reproduced with permission from Radford et al. (2015).

Microsoft Exhibit 1043 - Page 174
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985

the distinction between male and female, or about the presence or absence of glasses, without having to characterize all the configurations of the $n-1$ other features by examples covering all these combinations of values. This form of statistical separability is what allows one to generalize to new configurations of a person's features that have never been seen during training.

15.5 Exponential Gains from Depth

We have seen in section 6.4.1 that multilayer perceptrons are universal approximators, and that some functions can be represented by exponentially smaller deep networks compared to shallow networks. This decrease in model size leads to improved statistical efficiency. In this section, we describe how similar results apply more generally to other kinds of models with distributed hidden representations.

In section 15.4, we saw an example of a generative model that learned about the explanatory factors underlying images of faces, including the person's gender and whether they are wearing glasses. The generative model that accomplished this task was based on a deep neural network. It would not be reasonable to expect a shallow network, such as a linear network, to learn the complicated relationship between these abstract explanatory factors and the pixels in the image. In this and other AI tasks, the factors that can be chosen almost independently from each other yet still correspond to meaningful inputs are more likely to be very high level and related in highly nonlinear ways to the input. We argue that this demands *deep* distributed representations, where the higher level features (seen as functions of the input) or factors (seen as generative causes) are obtained through the composition of many nonlinearities.

It has been proved in many different settings that organizing computation through the composition of many nonlinearities and a hierarchy of reused features can give an exponential boost to statistical efficiency, on top of the exponential boost given by using a distributed representation. Many kinds of networks (e.g., with saturating nonlinearities, Boolean gates, sum/products, or RBF units) with a single hidden layer can be shown to be universal approximators. A model family that is a universal approximator can approximate a large class of functions (including all continuous functions) up to any nonzero tolerance level, given enough hidden units. However, the required number of hidden units may be very large. Theoretical results concerning the expressive power of deep architectures state that there are families of functions that can be represented efficiently by an architecture of depth k, but that would require an exponential number of hidden units (with respect to the input size) with insufficient depth (depth 2 or depth $k-1$).

Microsoft Exhibit 1043 - Page 175
Microsoft Corporation v. RealD Spark, LLC
IPR2023-01046 - U.S. Patent No. 10,740,985