Exhibit C13 – Invalidity of U.S. Patent No. RE49,461 – nnet

Prior Art Reference: Matlab Neural Network Toolbox User's Guide Version 4 ("nnet")

nnet was published in October 2004 and therefore qualifies as prior art to RE49,461 (the "'461 patent") under, at least, (pre-AIA) 35 U.S.C. § 102(a) and/or § 102(b). nnet anticipates and/or renders obvious claims 21–31 of the '461 patent (the "Asserted Claims").

The chart below provides representative examples of where each element of each claim is found within nnet. Citations are meant to be exemplary, not exhaustive, and NVIDIA reserves the right to identify and discuss additional portions of the reference in support of its contentions and/or to rebut arguments made by Plaintiff. Where NVIDIA states that nnet "discloses" a limitation, that disclosure may be express, implicit, and/or inherent. Citations to figures, drawings, tables, and the like include reference to any accompanying or related text. All internal cross-references are meant to incorporate the cross-referenced material as if fully set forth therein.

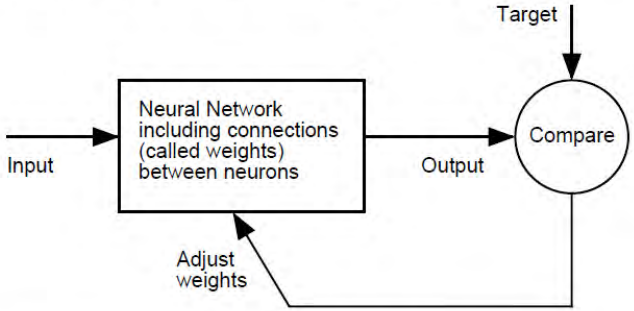
It is NVIDIA's position that Plaintiff's Infringement Contentions have not established that any accused product or service infringes any valid claim. Thus, NVIDIA's statements below should not be treated as an admission, implication, or suggestion that NVIDIA agrees with Plaintiff regarding either the scope, construction, or interpretation of any of the Asserted Claims or the infringement theories advanced by Plaintiff in its Infringement Contentions, including whether any Asserted Claim satisfies 35 U.S.C. §§ 101 or 112. The exemplary disclosures in these invalidity contentions are informed by the apparent interpretation of the asserted claims by Plaintiff. These statements are not intended to suggest that NVIDIA agrees with Plaintiff's application of any claim term, suggest a proposed construction at this stage of the case, or suggest that construction is needed.

To the extent Plaintiff argues that any element below is not disclosed by nnet, a person of ordinary skill in the art would have found it obvious to combine the teachings of nnet with the background knowledge of a person of ordinary skill in the art and/or the additional references, and exemplary teachings, set forth in NVIDIA's Invalidity Contentions, including all exhibits and appendices thereto. Additional exemplary combinations and reasons for obviousness are described in the cover pleading, and are incorporated by reference herein.

Plaintiff has yet to identify any limitation of the Asserted Claims that it contends is not anticipated and/or rendered obvious by nnet. NVIDIA therefore expressly reserves the right to respond to any such contention, including by identifying additional obviousness combinations, if Plaintiff makes any such contention. NVIDIA reserves the right to rely on additional citations or sources of evidence that also may be applicable, or that may become applicable in light of claim construction, changes in Plaintiff's infringement contentions, and/or information obtained during discovery as the case progresses.

Neural AI, LLC, Exhibit 2015
Page 2015 - 1
IPR2025-00608, NVIDIA Corporation v. Neural AI, LLC

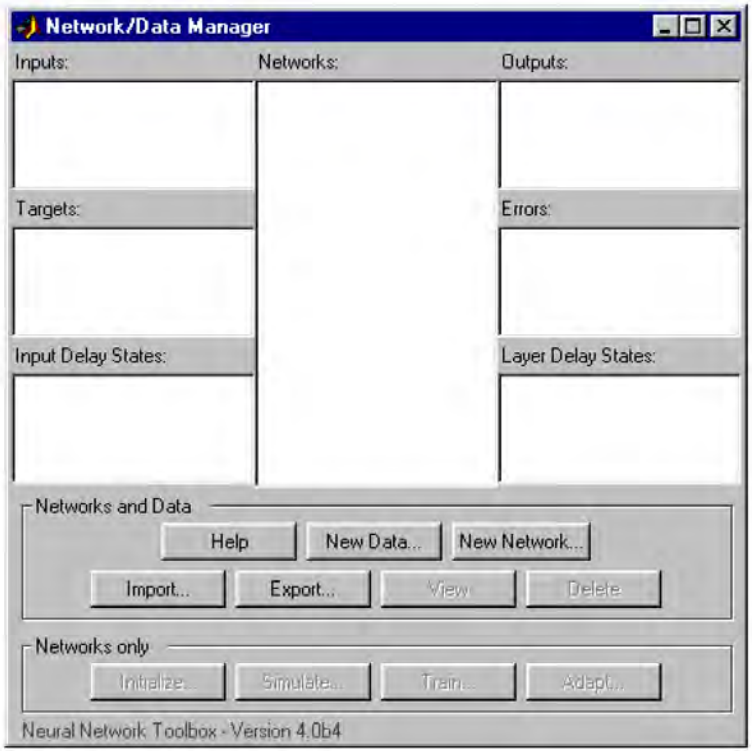
U.S. Patent No. RE49,461	nnet
21[pre] A method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition, the method comprising:	NVIDIA takes no position in these Invalidity Contentions on whether the entirety of the preamble of this claim is a limitation. To the extent it is a limitation, nnet discloses, expressly and/or inherently a method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition. *See* Chapters 2-8, 10-14. *See e.g.*: *As a non-limiting example, nnet discloses a method of executing computations representing an artificial neural network on a computer system (interactive GUI through which a user can define, set, or edit functions, parameters, variables, and sources of inputs to an artificial neural network, such as commands and data e.g., 392-413).*

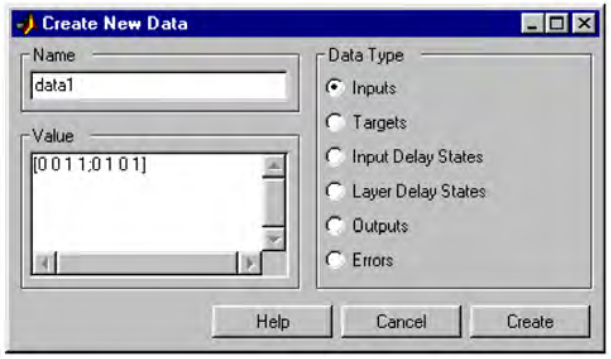
U.S. Patent No. RE49,461	nnet
6.	**Neural Networks** Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements. Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown below. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this *supervised learning*, to train a network.  Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as "on line" or "adaptive" training. Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. A list of applications is given in Chapter 1. Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. Throughout the toolbox emphasis is placed on neural network paradigms that build up to or are themselves used in engineering, financial and other practical applications.

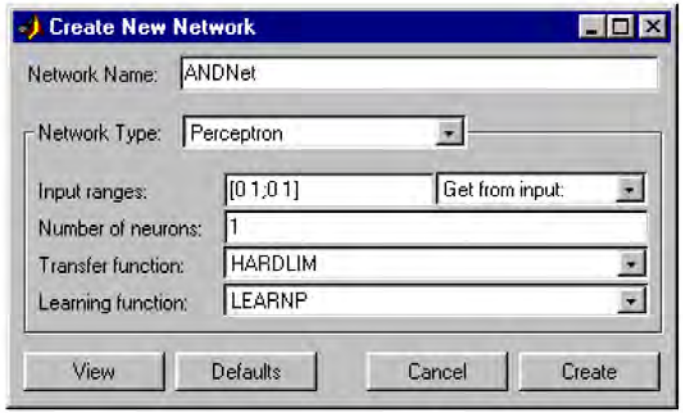
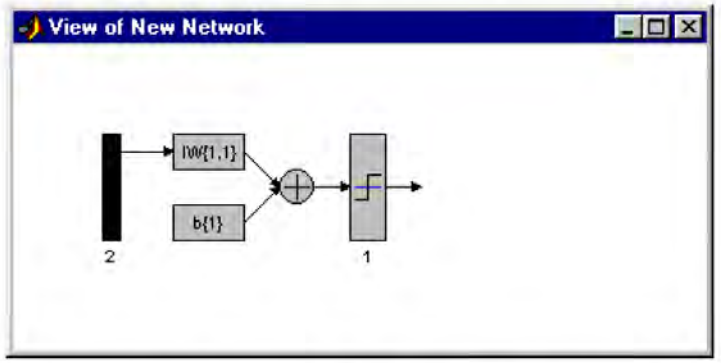
U.S. Patent No. RE49,461	nnet
	Preface **Neural Network Design Book** Professor Martin Hagan of Oklahoma State University, and Neural Network Toolbox authors Howard Demuth and Mark Beale have written a textbook, *Neural Network Design* (ISBN 0-9717321-0-8). The book presents the theory of neural networks, discusses their design and application, and makes considerable use of MATLAB and the Neural Network Toolbox. Demonstration programs from the book are used in various chapters of this Guide. (You can find all the book demonstration programs in the Neural Network Toolbox by typing nnd.) The book has: • An INSTRUCTOR'S MANUAL for adopters and • TRANSPARENCY OVERHEADS for class use. This book can be obtained from the University of Colorado Bookstore at 1-303-492-3648 or at the online purchase web site, cubooks.colorado.edu. To obtain a copy of the INSTRUCTOR'S MANUAL contact the University of Colorado Bookstore phone 1-303-492-3648. Ask specifically for an instructor's manual if you are instructing a class and want one. You can go directly to the Neural Network Design page at http://ee.okstate.edu/mhagan/nnd.html Once there, you can download the TRANSPARENCY MASTERS with a click on "Transparency Masters(3.6MB)". You can get the Transparency Masters in Powerpoint or PDF format. You can obtain sample book chapters in PDF format as well. 12.

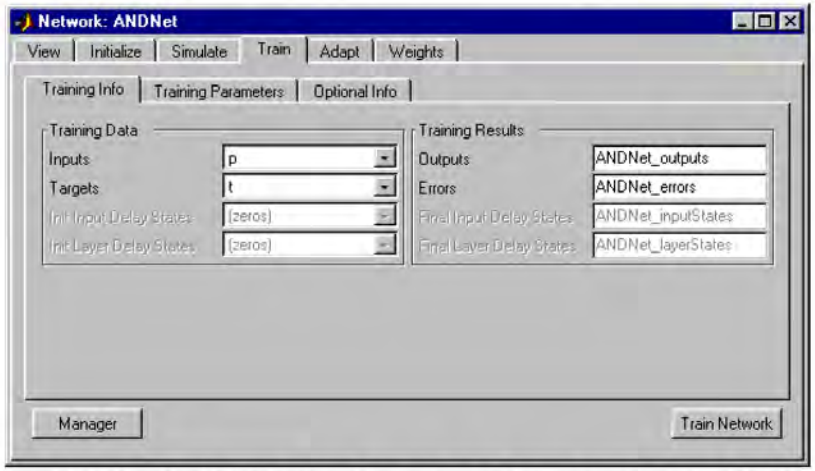
U.S. Patent No. RE49,461	nnet
	  What's New in Version 4.0 A few of the new features and improvements introduced with this version of the Neural Network Toolbox are discussed below. **Control System Applications** A new Chapter 6 presents three practical control systems applications: • Network model predictive control • Model reference adaptive control • Feedback linearization controller **Graphical User Interface** A graphical user interface has been added to the toolbox. This interface allows you to: • Create networks • Enter data into the GUI • Initialize, train, and simulate networks • Export the training results from the GUI to the command line workspace • Import data from the command line workspace to the GUI To open the Network/Data Manager window type `nntool`. **New Training Functions** The toolbox now has four training algorithms that apply weight and bias learning rules. One algorithm applies the learning rules in batch mode. Three algorithms apply learning rules in three different incremental modes: • `trainb` - Batch training function • `trainc` - Cyclical order incremental training function • `trainr` - Random order incremental training function • `trains` - Sequential order incremental training function All four functions present the whole training set in each epoch (pass through the entire input set). 31.

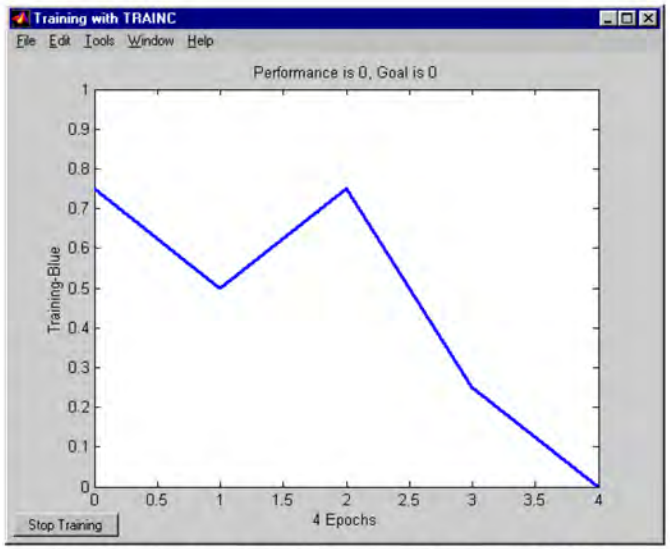
U.S. Patent No. RE49,461	nnet
91.	# Graphical User Interface ## Introduction to the GUI The graphical user interface (GUI) is designed to be simple and user friendly, but we will go through a simple example to get you started. In what follows you bring up a GUI **Network/Data Manager** window. This window has its own work area, separate from the more familiar command line workspace. Thus, when using the GUI, you might "export" the GUI results to the (command line) workspace. Similarly you may want to "import" results from the command line workspace to the GUI. Once the **Network/Data Manager** is up and running, you can create a network, view it, train it, simulate it and export the final results to the workspace. Similarly, you can import data from the workspace for use in the GUI. The following example deals with a perceptron network. We go through all the steps of creating a network and show you what you might expect to see as you go along. ## Create a Perceptron Network (nntool) We create a perceptron network to perform the AND function in this example. It has an input vector `p= [0 0 1 1;0 1 0 1]` and a target vector `t=[0 0 0 1]`. We call the network `ANDNet`. Once created, the network will be trained. We can then save the network, its output, etc., by "exporting" it to the command line. ### Input and target To start, type `nntool`. The following window appears.

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value `[0 0 0 1]`, and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter `ANDNet` under **Network Name**. Set the **Network Type** to `Perceptron`, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges `[0 1;0 1]`.We want to use a `hardlim` transfer function and a `learnp` learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

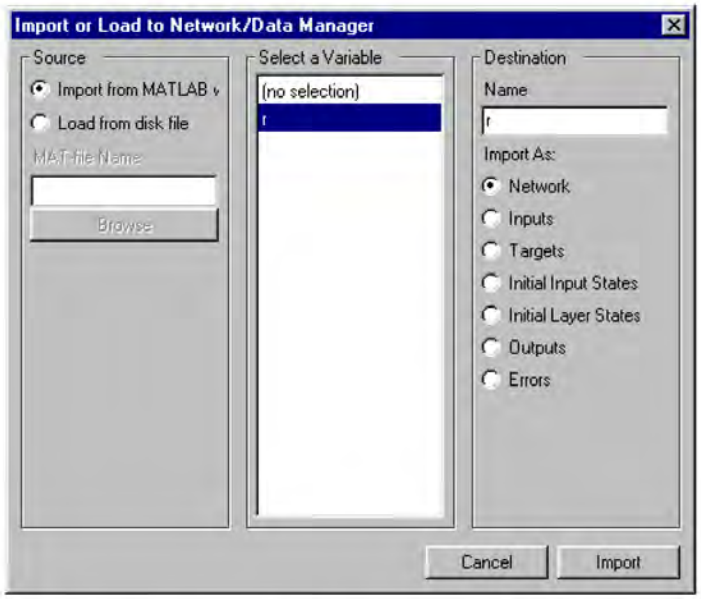
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on ANDNet to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting p from the pop-down list of inputs and t from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input p and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify p as the input, and label the output as ANDNet_outputsSim to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: ANDNet_outputsSim. 96.

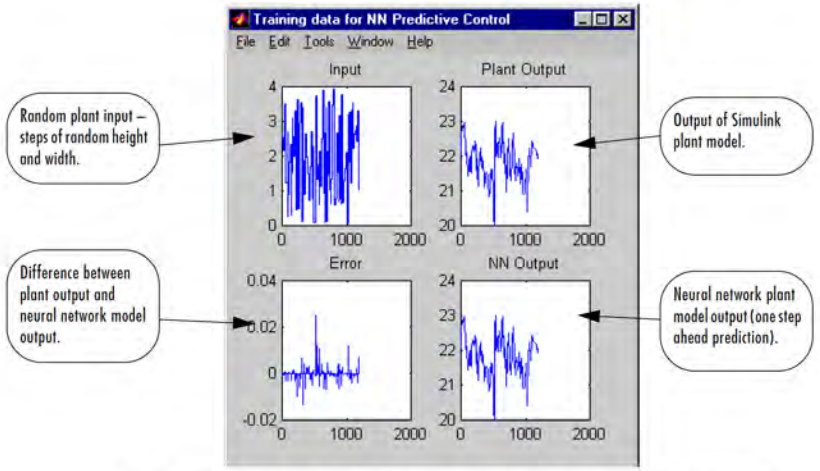
U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
218.	 according to the selected training algorithm (`trainlm` in this case). This is a straightforward application of batch training, as described in Chapter 5, "Backpropagation." After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.) You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this demonstration, begin the simulation, as shown in the following steps.  **7** Select **OK** in the **Plant Identification** window. This loads the trained neural network plant model into the NN Predictive Controller block. **8** Select **OK** in the **Neural Network Predictive Control** window. This loads the controller parameters into the NN Predictive Controller block. **9** Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the

U.S. Patent No. RE49,461	nnet
	237.

Importing and Exporting

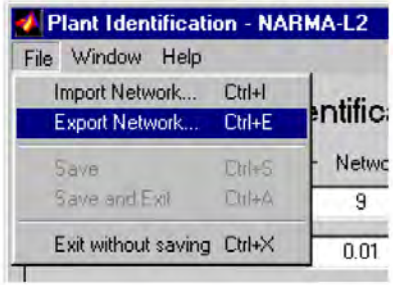
You can save networks and training data to the workspace or to a disk file. The following two sections demonstrate how you can do this.

Importing and Exporting Networks

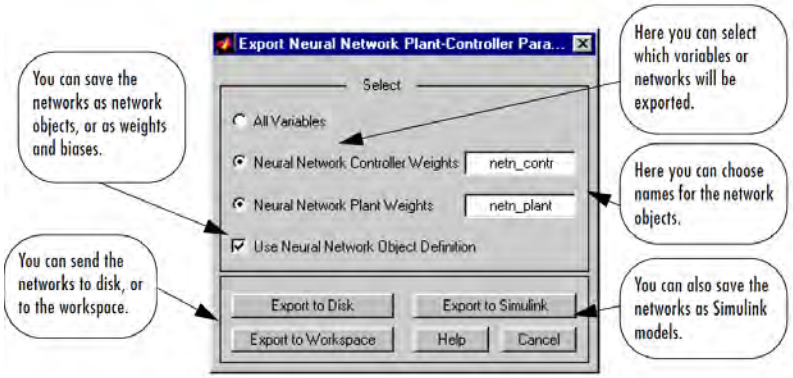
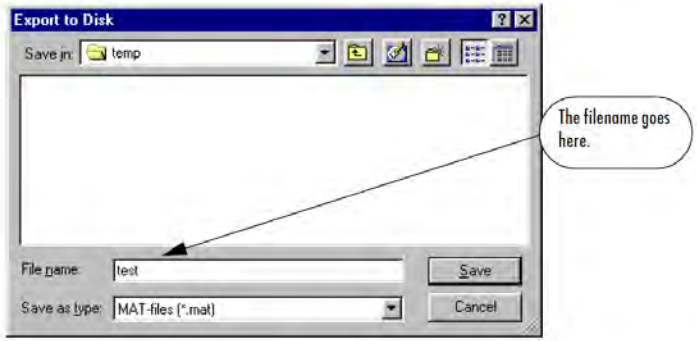
The controller and plant model networks that you develop are stored within Simulink controller blocks. At some point you may want to transfer the networks into other applications, or you may want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following demonstration leads you through the export and import processes. (We use the NARMA-L2 window for this demonstration, but the same procedure applies to all of the controllers.)

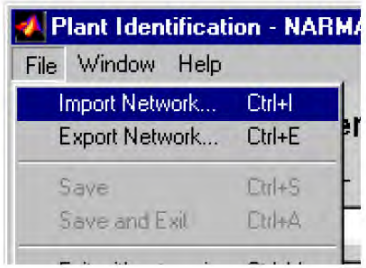
1 Repeat the first three steps of the NARMA-L2 demonstration. The **NARMA-L2 Plant Identification** window should then be open.

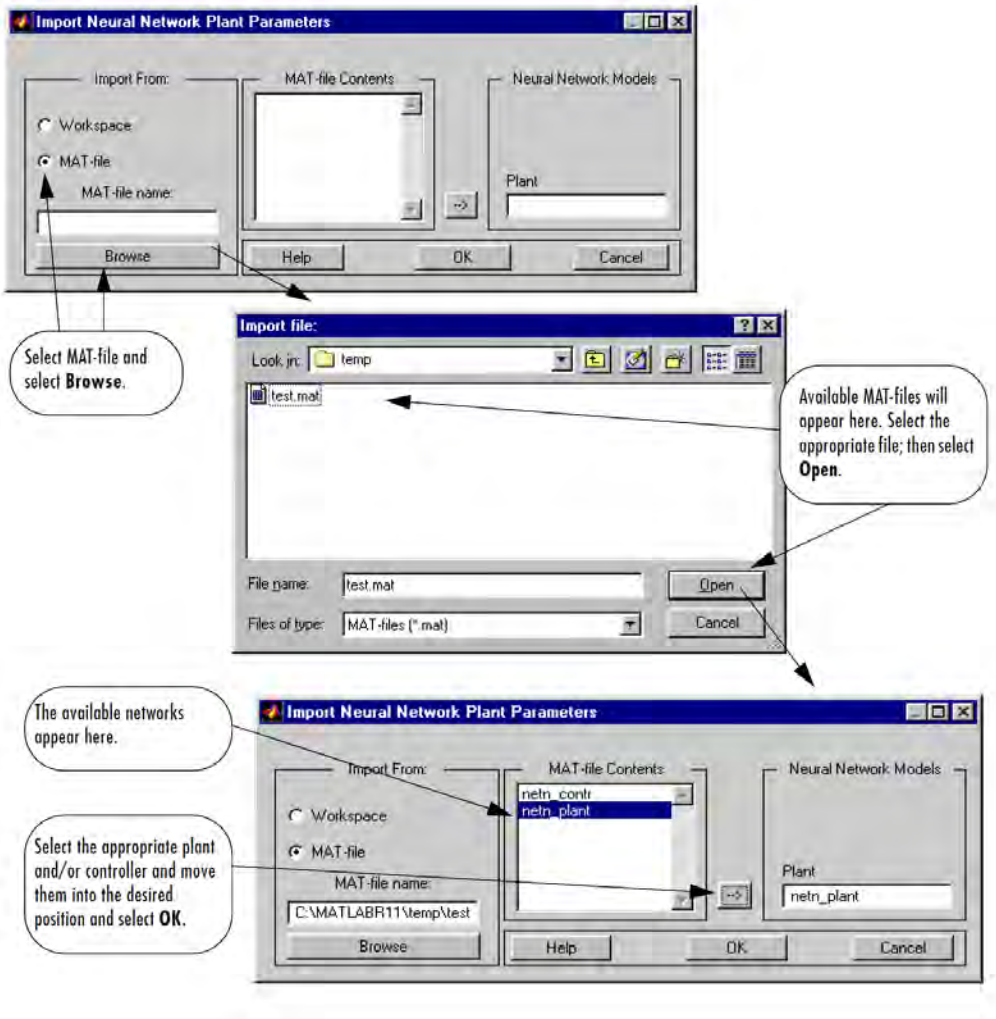
2 Select **Export** from the **File** menu, as shown below.

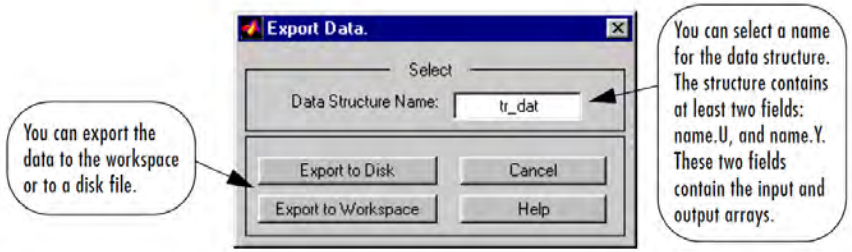


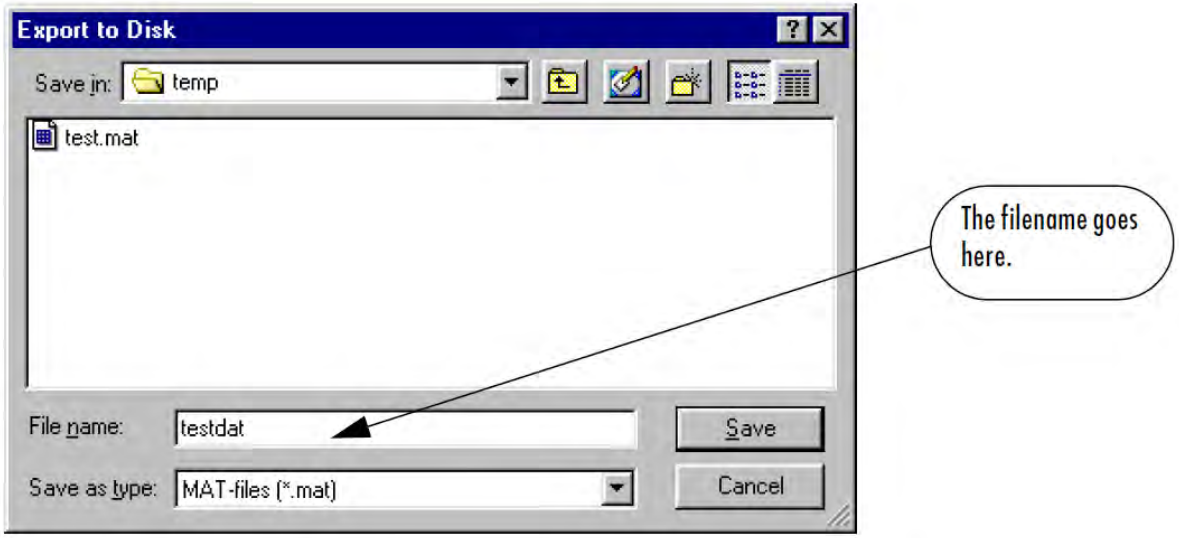
This causes the following window to open.

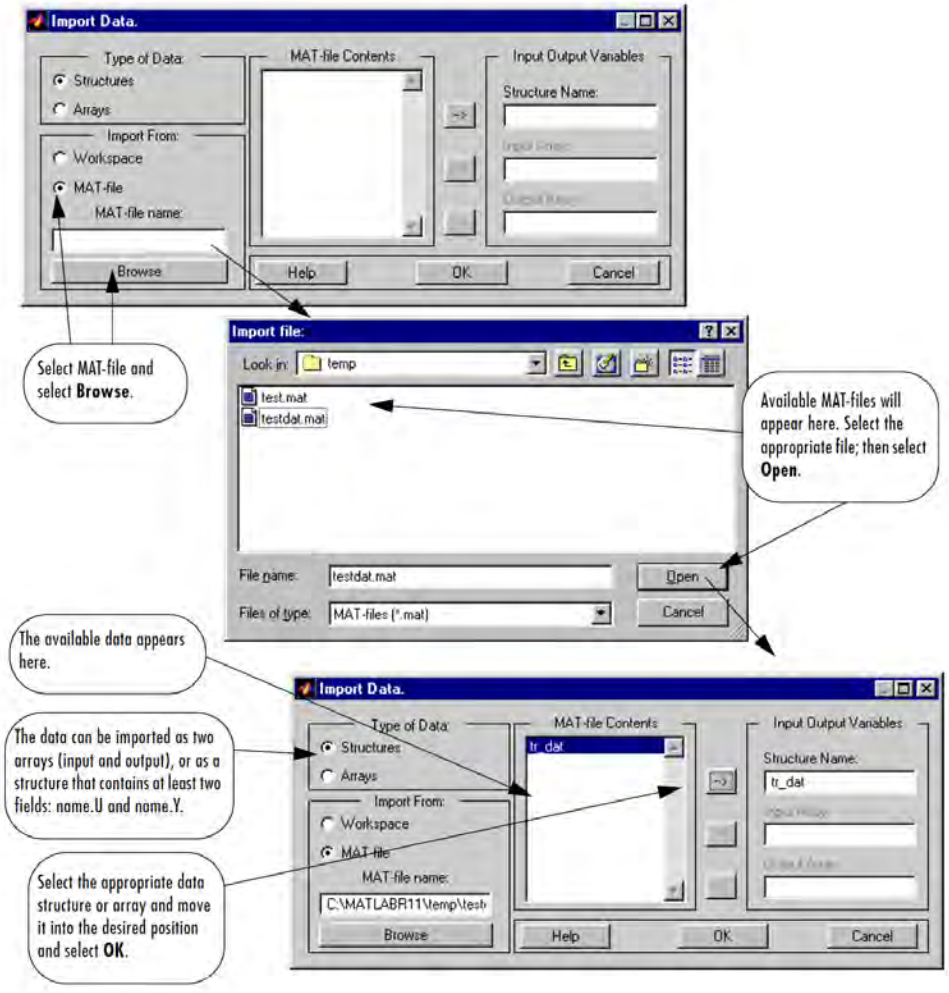
U.S. Patent No. RE49,461	nnet
	 **3** Select **Export to Disk**. The following window opens. Enter the filename `test` in the box, and select **Save**. This saves the controller and plant networks to disk.  **4** Retrieve that data with the **Import** menu option. Select **Import Network** from the **File** menu, as in the following figure. 238.

U.S. Patent No. RE49,461	nnet
	 This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is retrieved, you can load it into the controller block by selecting **OK** or **Apply**. Notice that the window only has an entry for the plant model, even though you saved both the plant model and the controller. This is because the NARMA-L2 controller is derived directly from the plant model, so you don't need to import both networks. 239.

U.S. Patent No. RE49,461	nnet
	 240.

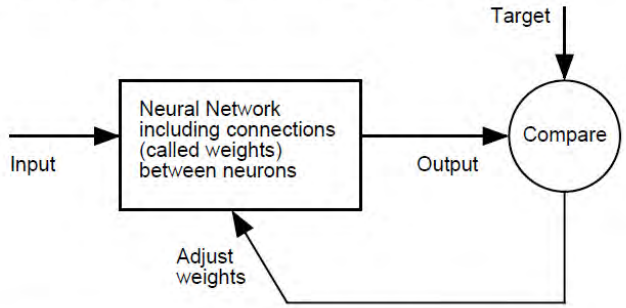
U.S. Patent No. RE49,461	nnet
	Importing and Exporting Training Data The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You may wish to save the training data to the workspace or to a disk file so that you can load it again at a later time. You may also want to combine data sets manually and then load them back into the training window. You can do this by using the Import and Export buttons. The following demonstration leads you through the import and export processes. (We use the **NN Predictive Control** window for this demonstration, but the same procedure applies to all of the controllers.) **1** Repeat the first five steps of the NN Predictive Control demonstration. Then select **Accept Data**. The **Plant Identification** window should then be open, and the Import and Export buttons should be active. **2** Select the **Export** button. This causes the following window to open.  **3** Select **Export to Disk**. The following window opens. Enter the filename `testdat` in the box, and select **Save**. This saves the training data structure to disk. 241.

U.S. Patent No. RE49,461	nnet
	 **4** Now let's retrieve the data with the import command. Select the **Import** button in the **Plant Identification** window.This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model. 242.

U.S. Patent No. RE49,461	nnet
	 243.

U.S. Patent No. RE49,461	nnet
	6 Control Systems **Summary** The following table summarizes the controllers discussed in this chapter. **Block** / **Description** NN Predictive Control — Uses a neural network plant model to predict future plant behavior. An optimization algorithm determines the control input that optimizes plant performance over a finite time horizon. The plant training requires only a batch algorithm for static networks and is reasonably fast. The controller requires an online optimization algorithm, which requires more computation than the other controllers. NARMA-L2 Control — An approximate plant model is in companion form. The next control input is computed to force the plant output to follow a reference signal. The neural network plant model is trained with static backpropagation and is reasonably fast. The controller is a rearrangement of the plant model, and requires minimal online computation. Model Reference Control — A neural network plant model is first developed. The plant model is then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a more general class of plant than does the NARMA-L2 control architecture. The controller requires minimal online computation. 244. To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the

U.S. Patent No. RE49,461	nnet
	teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[a] executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition;	nnet discloses, expressly and/or inherently, executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition. *See* Chapters 2, 6. *See e.g.*: *As a non-limiting example, nnet discloses at least one CPU (e.g., Matlab is executable on a computer), and a user interaction stream (e.g., GUI) controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition.*

U.S. Patent No. RE49,461	nnet
6.	**Neural Networks** Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements. Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown below. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this *supervised learning*, to train a network.  Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as "on line" or "adaptive" training. Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. A list of applications is given in Chapter 1. Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. Throughout the toolbox emphasis is placed on neural network paradigms that build up to or are themselves used in engineering, financial and other practical applications.

U.S. Patent No. RE49,461	nnet
	60.

2 Neuron Model and Network Architectures

Summary

The inputs to a neuron include its bias and the sum of its weighted inputs (using the inner product). The output of a neuron depends on the neuron's inputs and on its transfer function. There are many useful transfer functions.

A single neuron cannot do very much. However, several neurons can be combined into a layer or multiple layers that have great power. Hopefully this toolbox makes it easy to create and understand such large networks.

The architecture of a network consists of a description of how many layers a network has, the number of neurons in each layer, each layer's transfer function, and how the layers connect to each other. The best architecture to use depends on the type of problem to be represented by the network.

A network effects a computation by mapping input values to output values. The particular mapping problem to be performed fixes the number of inputs, as well as the number of outputs for the network.

Aside from the number of neurons in a network's output layer, the number of neurons in each layer is up to the designer. Except for purely linear networks, the more neurons in a hidden layer, the more powerful the network.

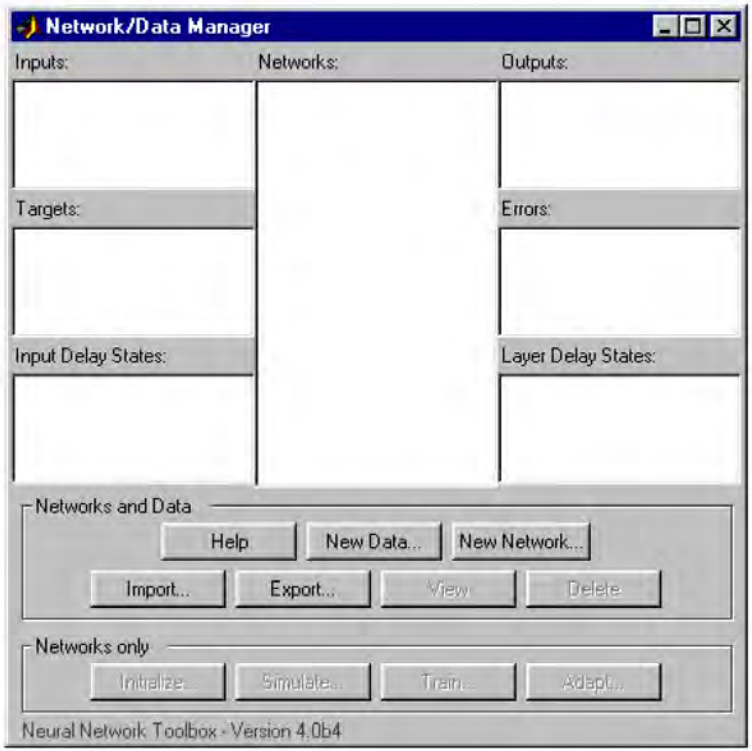
If a linear mapping needs to be represented linear neurons should be used. However, linear networks cannot perform any nonlinear computation. Use of a nonlinear transfer function makes a network capable of storing nonlinear relationships between input and output.

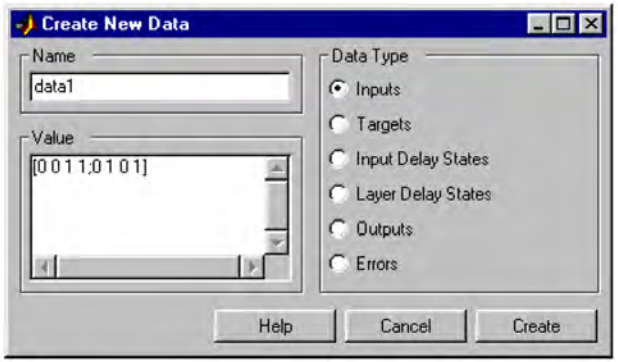
A very simple problem can be represented by a single layer of neurons. However, single-layer networks cannot solve certain problems. Multiple feed-forward layers give a network greater freedom. For example, any reasonable function can be represented with a two-layer network: a sigmoid layer feeding a linear output layer.

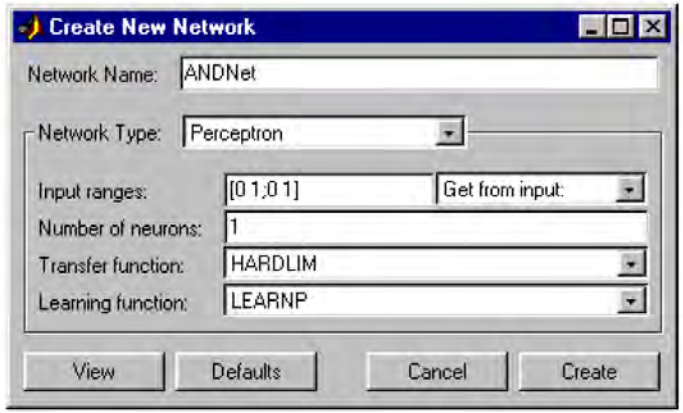
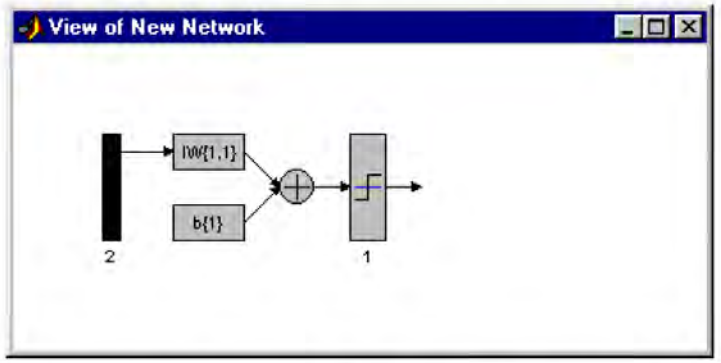
Networks with biases can represent relationships between inputs and outputs more easily than networks without biases. (For example, a neuron without a bias will always have a net input to the transfer function of zero when all of its inputs are zero. However, a neuron with a bias can learn to have any net transfer function input under the same conditions by learning an appropriate value for the bias.)

Feed-forward networks cannot perform temporal computation. More complex networks with internal feedback paths are required for temporal behavior.

U.S. Patent No. RE49,461	nnet
	# Graphical User Interface ## Introduction to the GUI The graphical user interface (GUI) is designed to be simple and user friendly, but we will go through a simple example to get you started. In what follows you bring up a GUI **Network/Data Manager** window. This window has its own work area, separate from the more familiar command line workspace. Thus, when using the GUI, you might "export" the GUI results to the (command line) workspace. Similarly you may want to "import" results from the command line workspace to the GUI. Once the **Network/Data Manager** is up and running, you can create a network, view it, train it, simulate it and export the final results to the workspace. Similarly, you can import data from the workspace for use in the GUI. The following example deals with a perceptron network. We go through all the steps of creating a network and show you what you might expect to see as you go along. ## Create a Perceptron Network (nntool) We create a perceptron network to perform the AND function in this example. It has an input vector `p= [0 0 1 1;0 1 0 1]` and a target vector `t=[0 0 0 1]`. We call the network `ANDNet`. Once created, the network will be trained. We can then save the network, its output, etc., by "exporting" it to the command line. ### Input and target To start, type `nntool`. The following window appears. 91.

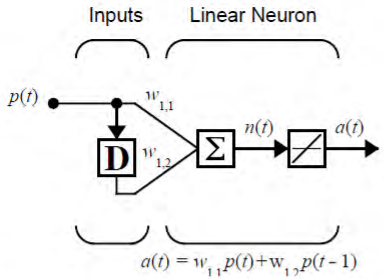
U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

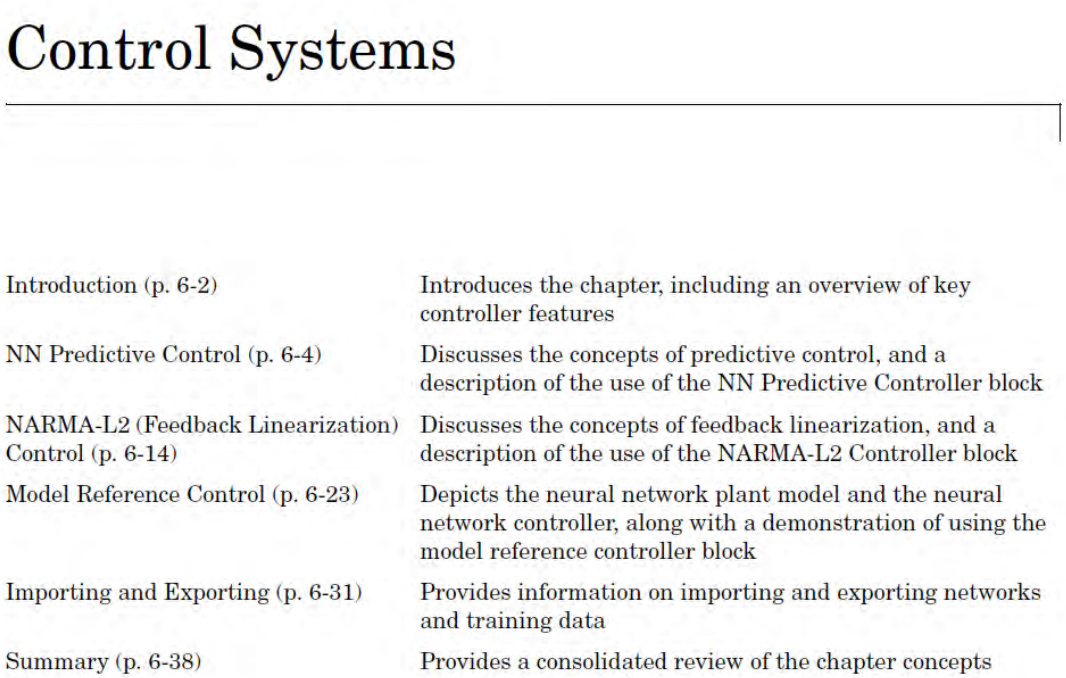
U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value `[0 0 0 1]`, and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter `ANDNet` under **Network Name**. Set the **Network Type** to `Perceptron`, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges `[0 1;0 1]`.We want to use a `hardlim` transfer function and a `learnp` learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

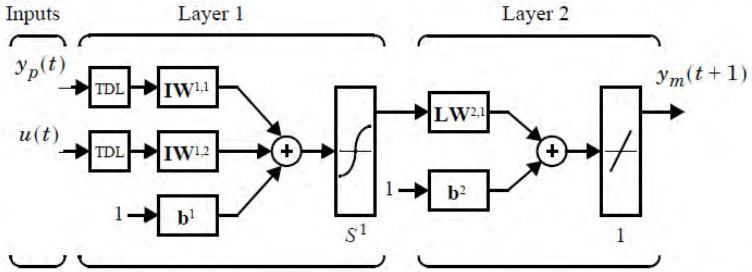
U.S. Patent No. RE49,461	nnet

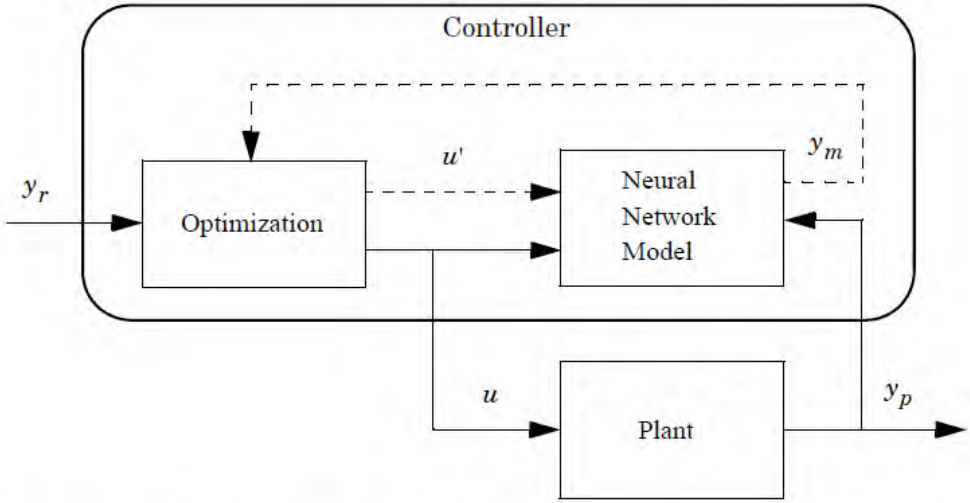
51.	
	The following commands create this network:
	`net = newlin([-1 1],1,[0 1]);`
	`net.biasConnect = 0;`
	Assign the weight matrix to be
	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$.
	The command is
	`net.IW{1,1} = [1 2];`
	Suppose that the input sequence is
	$\mathbf{p1} = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{p2} = \begin{bmatrix} 2 \end{bmatrix}$, $\mathbf{p3} = \begin{bmatrix} 3 \end{bmatrix}$, $\mathbf{p4} = \begin{bmatrix} 4 \end{bmatrix}$,
	Sequential inputs are presented to the network as elements of a cell array:
	`P = {1 2 3 4};`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`[1] [4] [7] [10]`
	We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case,

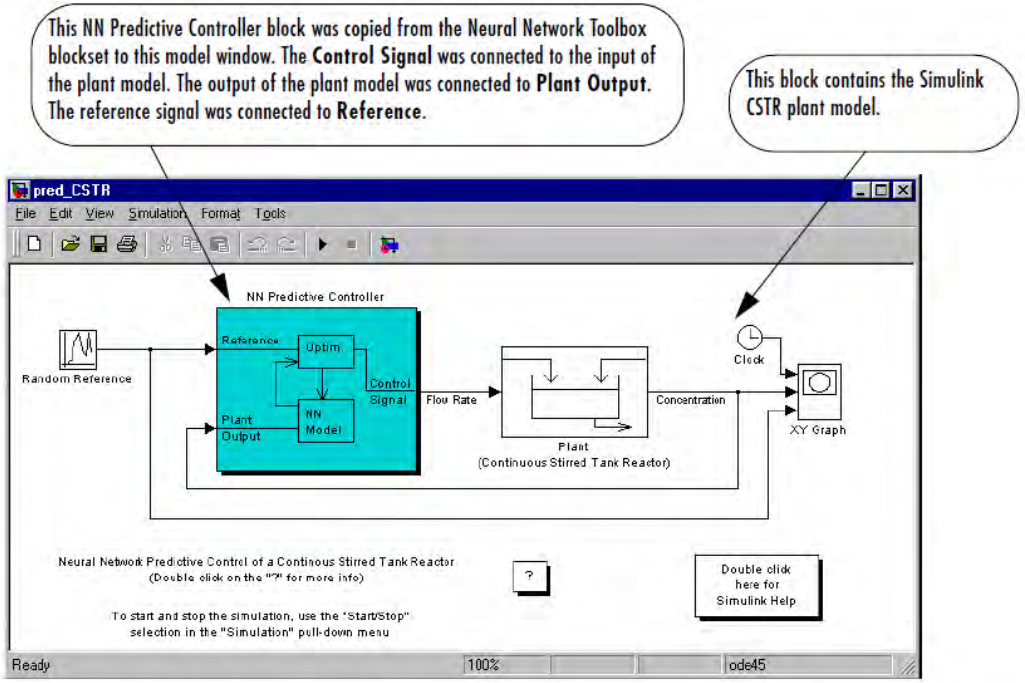
U.S. Patent No. RE49,461	nnet
	 207.

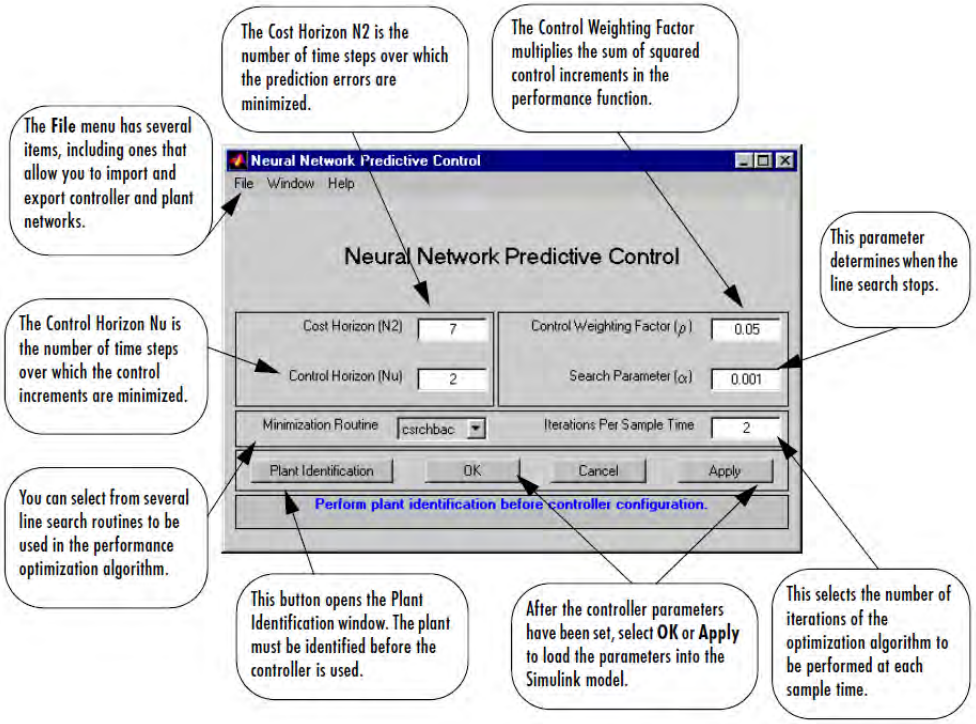
U.S. Patent No. RE49,461	nnet
	Introduction Neural networks have been applied very successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99]. This chapter introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox: • Model Predictive Control • NARMA-L2 (or Feedback Linearization) Control • Model Reference Control This chapter presents brief descriptions of each of these architectures and demonstrates how you can use them. There are typically two steps involved when using neural networks for control: **1** System Identification **2** Control Design In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this chapter, the system identification stage is identical. The control design stage, however, is different for each architecture. • For the model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance. • For the NARMA-L2 control, the controller is simply a rearrangement of the plant model. • For the model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training. The next three sections of this chapter discuss model predictive control, NARMA-L2 control and model reference control. Each section consists of a brief 208.

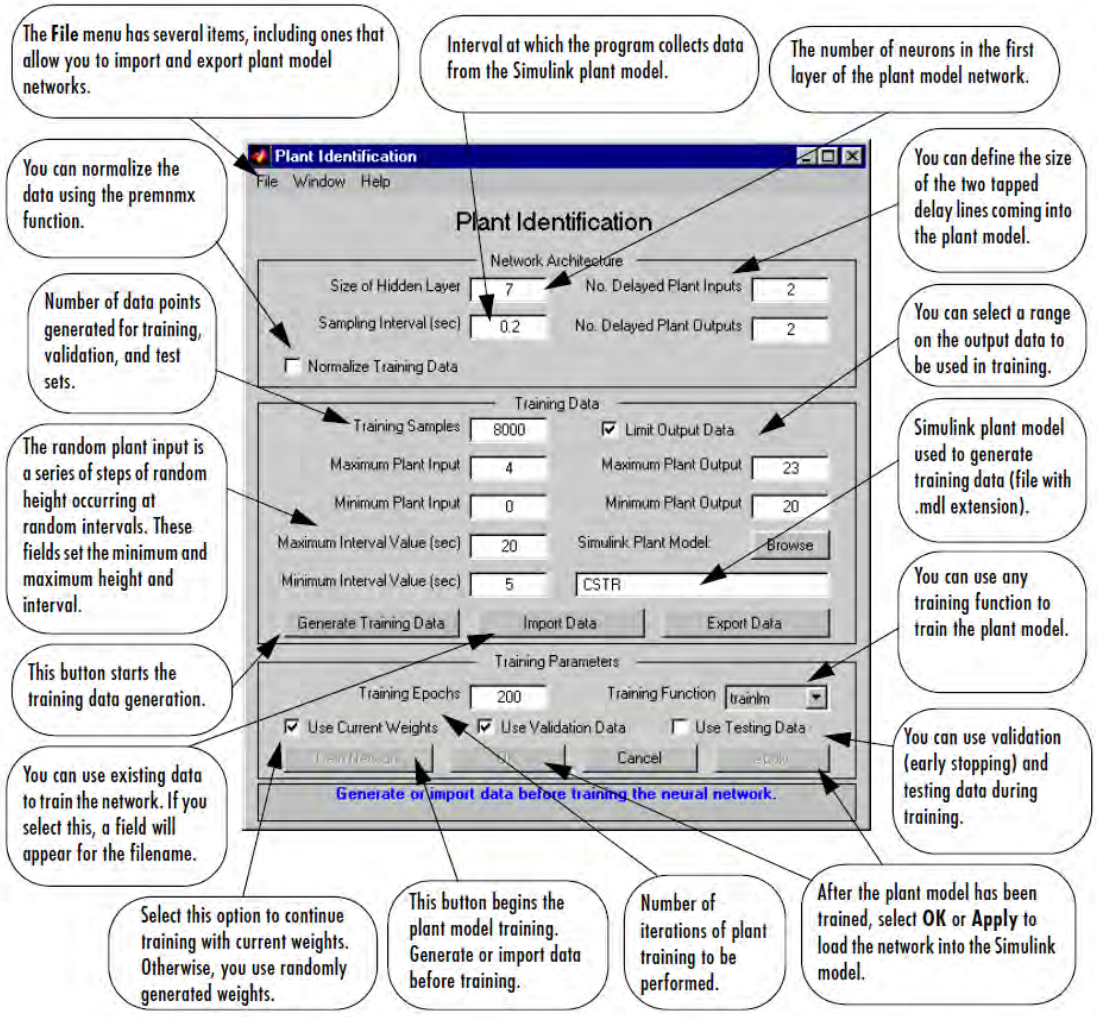
U.S. Patent No. RE49,461	nnet
	description of the control concept, followed by a demonstration of the use of the appropriate Neural Network Toolbox function. These three controllers are implemented as Simulink® blocks, which are contained in the Neural Network Toolbox blockset. To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application. • **Model Predictive Control**. This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form, using any of the training algorithms discussed in Chapter 5. (This is true for all three control architectures.) The controller, however, requires a significant amount of on-line computation, since an optimization algorithm is performed at each sample time to compute the optimal control input. • **NARMA-L2 Control**. This controller requires the least computation of the three architectures described in this chapter. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (The companion form model is described later in this chapter.) • **Model Reference Control**. The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained off-line, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation [HaJe99]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control. 209.

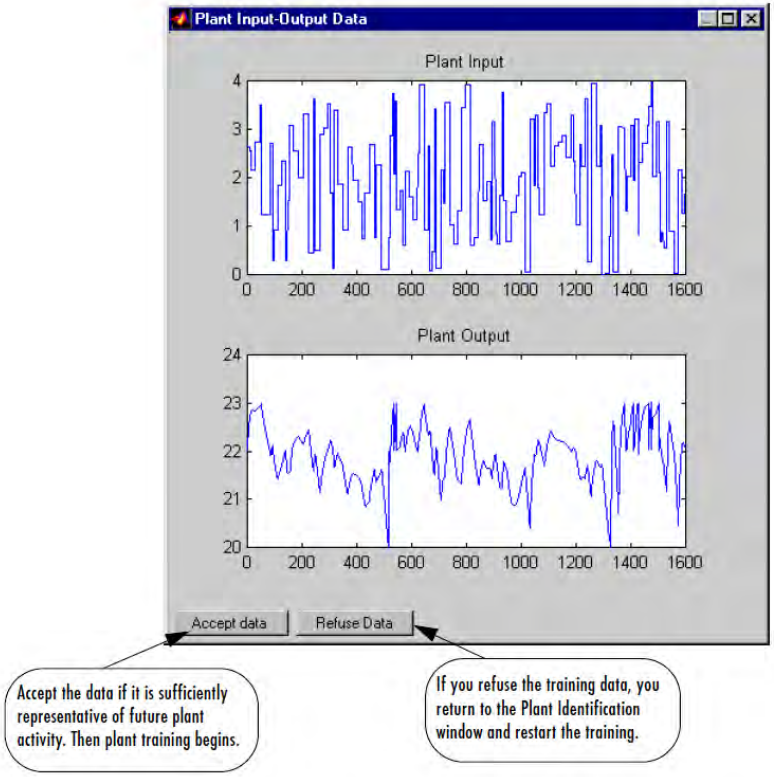
U.S. Patent No. RE49,461	nnet
	The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.  This network can be trained offline in batch mode, using data collected from the operation of the plant. Any of the training algorithms discussed in Chapter 5, "Backpropagation", can be used for network training. This process is discussed in more detail later in this chapter. **Predictive Control** The model predictive control method is based on the receding horizon technique [SoHa96]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon. $$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$ where N_1, N_2 and N_u define the horizons over which the tracking error and the control increments are evaluated. The u' variable is the tentative control signal, y_r is the desired response and y_m is the network model response. The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index. The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J, and 211.

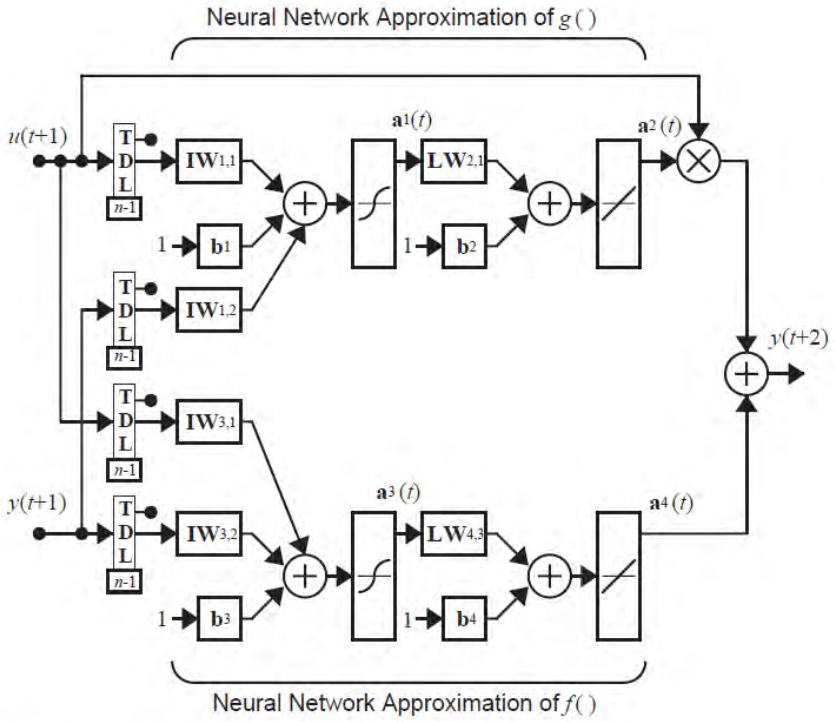
U.S. Patent No. RE49,461	nnet
	then the optimal u is input to the plant. The controller block has been implemented in Simulink, as described in the following section.  **Using the NN Predictive Controller Block** This section demonstrates how the NN Predictive Controller block is used.The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration. A demo model is provided with the Neural Network Toolbox to demonstrate the predictive controller. This demo uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure. 212.

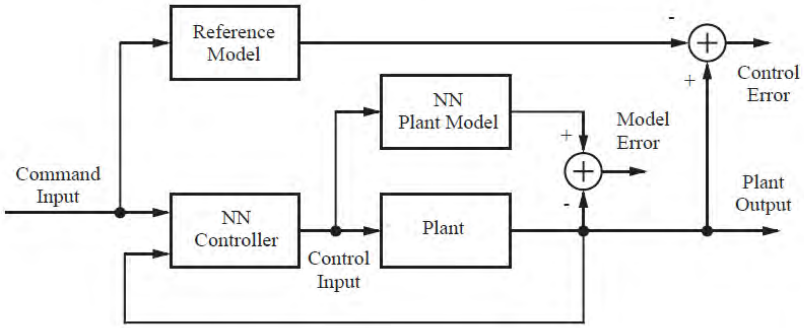
U.S. Patent No. RE49,461	nnet
214.	 **3** Double-click the NN Predictive Controller block. This brings up the following window for designing the model predictive controller. This window enables you to change the controller horizons N_2 and N_u. (N_1 is fixed at 1.) The weighting parameter ρ, described earlier, is also defined in this window. The parameter α is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in Chapter 5, "Backpropagation."

U.S. Patent No. RE49,461	nnet
	 **4** Select **Plant Identification**. This opens the following window. The neural network plant model must be developed before the controller is used. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. The size of that layer, the number of delayed inputs and delayed outputs, and the training function are selected in this window. You can select any of the training functions described in Chapter 5, "Backpropagation", to train the neural network plant model. 215.

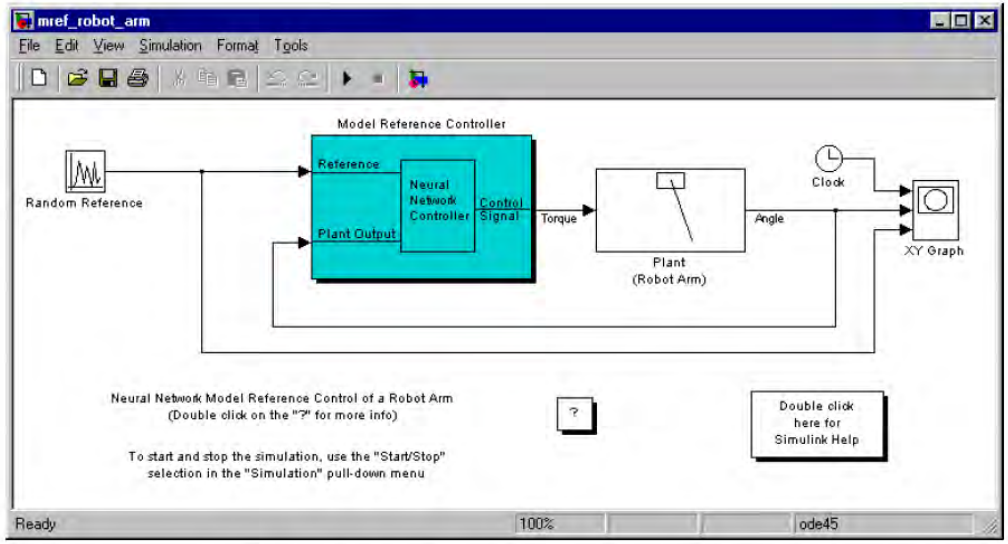
U.S. Patent No. RE49,461	nnet
	 216.

U.S. Patent No. RE49,461	nnet
	5 Select the **Generate Training Data** button. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.  **6** Select **Accept Data**, and then select **Train Network** from the Plant Identification window. Plant model training begins. The training proceeds 217.

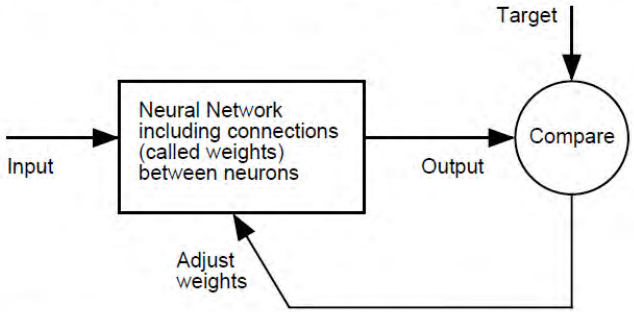
U.S. Patent No. RE49,461	nnet
	 **NARMA-L2 Controller** Using the NARMA-L2 model, you can obtain the controller $$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$ which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller. 222.

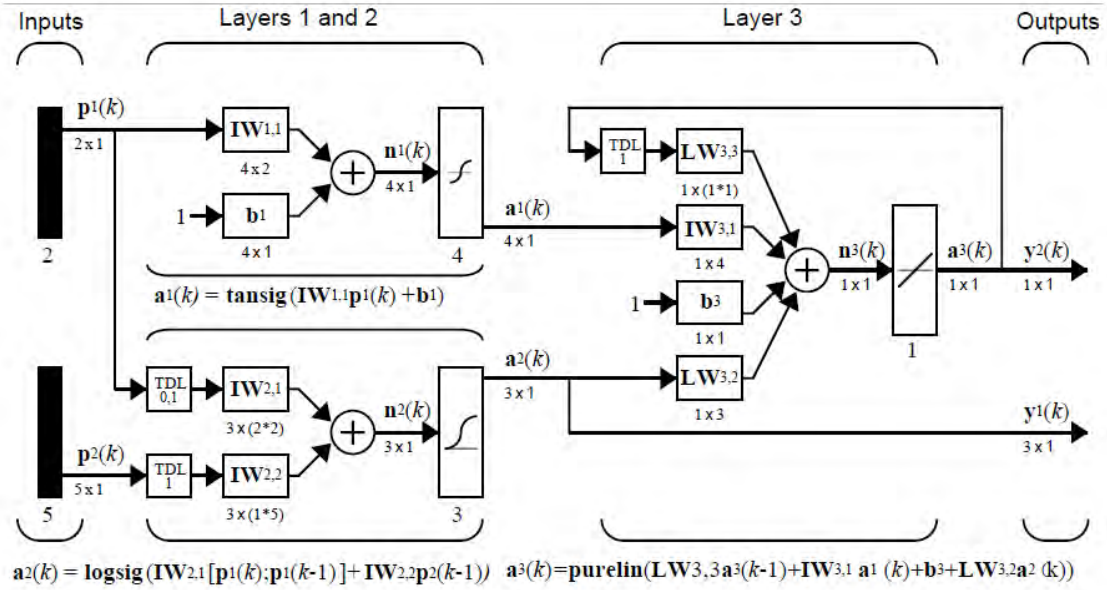
U.S. Patent No. RE49,461	nnet
	Model Reference Control The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.  The figure on the following page shows the details of the neural network plant model and the neural network controller, as they are implemented in the Neural Network Toolbox. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs: • Delayed reference inputs • Delayed controller outputs • Delayed plant outputs For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model: • Delayed controller outputs • Delayed plant outputs As with the controller, you can set the number of delays. The next section demonstrates how you can set the parameters. 229.

U.S. Patent No. RE49,461	nnet
	Summary The following table summarizes the controllers discussed in this chapter. **Block** / **Description** NN Predictive Control — Uses a neural network plant model to predict future plant behavior. An optimization algorithm determines the control input that optimizes plant performance over a finite time horizon. The plant training requires only a batch algorithm for static networks and is reasonably fast. The controller requires an online optimization algorithm, which requires more computation than the other controllers. NARMA-L2 Control — An approximate plant model is in companion form. The next control input is computed to force the plant output to follow a reference signal. The neural network plant model is trained with static backpropagation and is reasonably fast. The controller is a rearrangement of the plant model, and requires minimal online computation. Model Reference Control — A neural network plant model is first developed. The plant model is then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a more general class of plant than does the NARMA-L2 control architecture. The controller requires minimal online computation. 244.

U.S. Patent No. RE49,461	nnet
	This demo uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used. To run this demo, follow these steps. **1** Start MATLAB. **2** Run the demo model by typing `mrefrobotarm` in the MATLAB command window. This command starts Simulink and creates the following model window. The Model Reference Control block has already been placed in the model.  **3** Double-click the Model Reference Control block. This brings up the following window for training the model reference controller. 232.

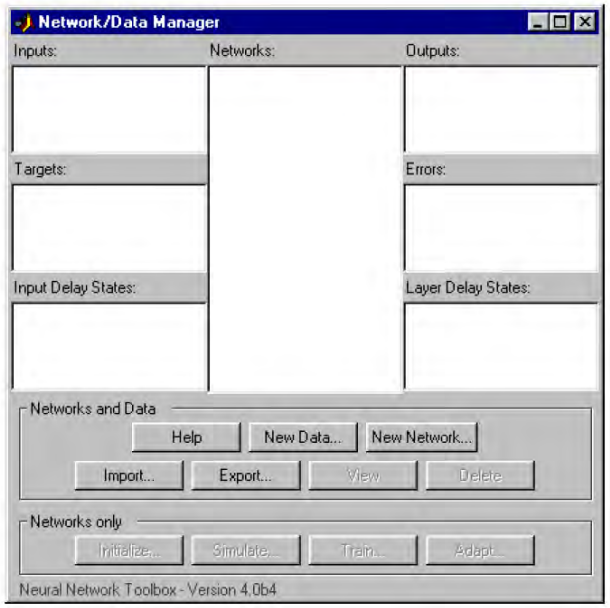
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[b] executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network;	nnet discloses, expressly and/or inherently, executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network. *See* Chapters 2-3, 5-6. *See e.g.*: *As a non-limiting example, nnet discloses executing by the processing unit, a computational stream (e.g., nntool) controlling the data exchange during execution of computations representing the artificial neural network.*

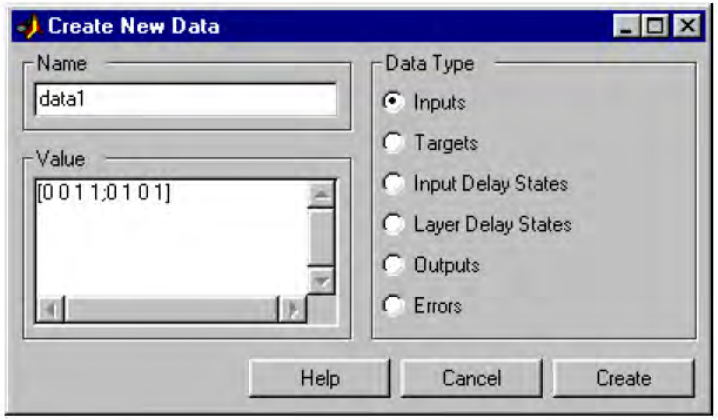
U.S. Patent No. RE49,461	nnet
6.	**Neural Networks** Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements. Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown below. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this *supervised learning*, to train a network.  Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as "on line" or "adaptive" training. Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. A list of applications is given in Chapter 1. Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. Throughout the toolbox emphasis is placed on neural network paradigms that build up to or are themselves used in engineering, financial and other practical applications.

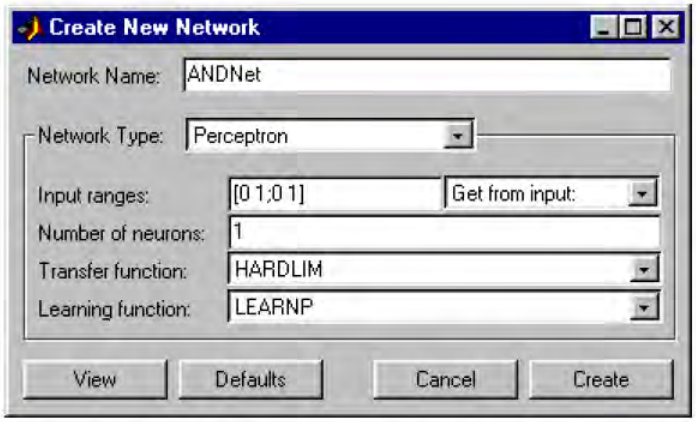
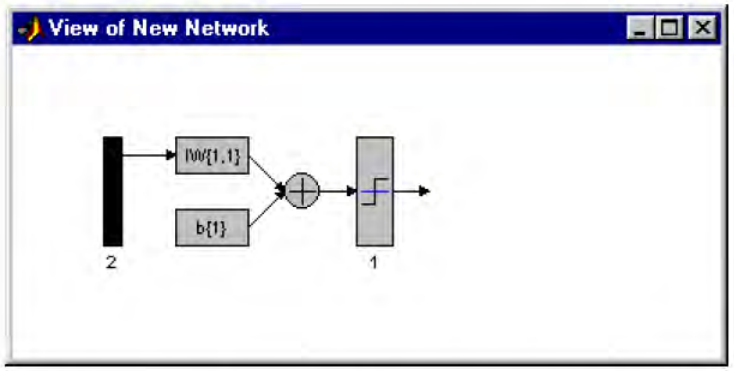
U.S. Patent No. RE49,461	nnet
	Input Weight Matrix $IW^{k,l}$ Layer Weight Matrix $LW^{k,l}$ **Figure and Equation Examples** The following figure, taken from Chapter 12 illustrates notation used in such advanced figures.  10.

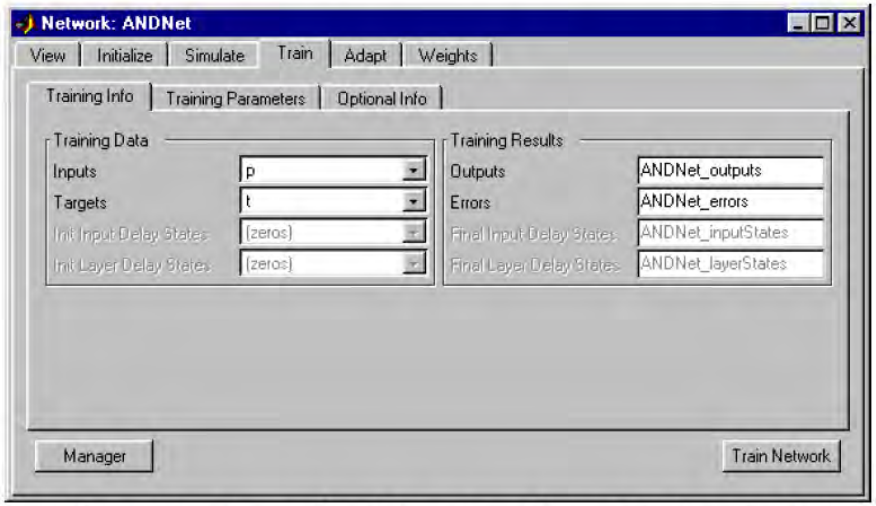
U.S. Patent No. RE49,461	nnet

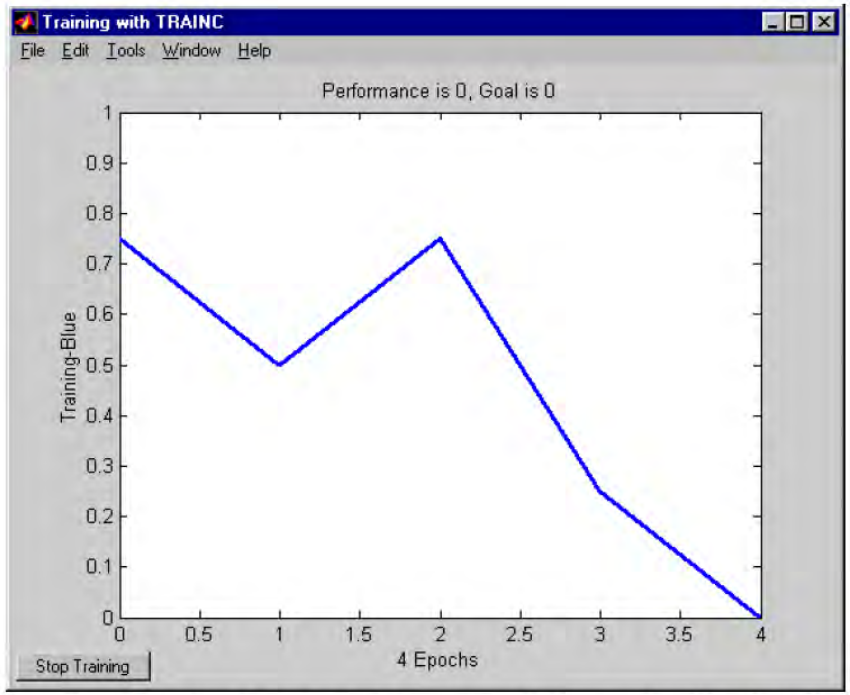
	Simulation With Concurrent Inputs in a Dynamic Network
	If we were to apply the same inputs from the previous example as a set of concurrent inputs instead of a sequence of inputs, we would obtain a completely different response. (Although, it is not clear why we would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, if we use a concurrent set of inputs we have
	$\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix}3\end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix}4\end{bmatrix}$
	which can be created with the following code:
	`P = [1 2 3 4];`
	When we simulate with concurrent inputs we obtain
	`A = sim(net,P)`
	`A =`
	`1 2 3 4`
	The result is the same as if we had concurrently applied each one of the inputs to a separate network and computed one output. Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero. For this case the output will simply be 1 times the input, since the weight that multiplies the current input is 1.
	In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences. For example, let's say we wanted to present the following two sequences to the network:
	$\mathbf{p}_1(1) = \begin{bmatrix}1\end{bmatrix}, \mathbf{p}_1(2) = \begin{bmatrix}2\end{bmatrix}, \mathbf{p}_1(3) = \begin{bmatrix}3\end{bmatrix}, \mathbf{p}_1(4) = \begin{bmatrix}4\end{bmatrix}$
	$\mathbf{p}_2(1) = \begin{bmatrix}4\end{bmatrix}, \mathbf{p}_2(2) = \begin{bmatrix}3\end{bmatrix}, \mathbf{p}_2(3) = \begin{bmatrix}2\end{bmatrix}, \mathbf{p}_2(4) = \begin{bmatrix}1\end{bmatrix}$
	52.

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
  2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
  ans =
  -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

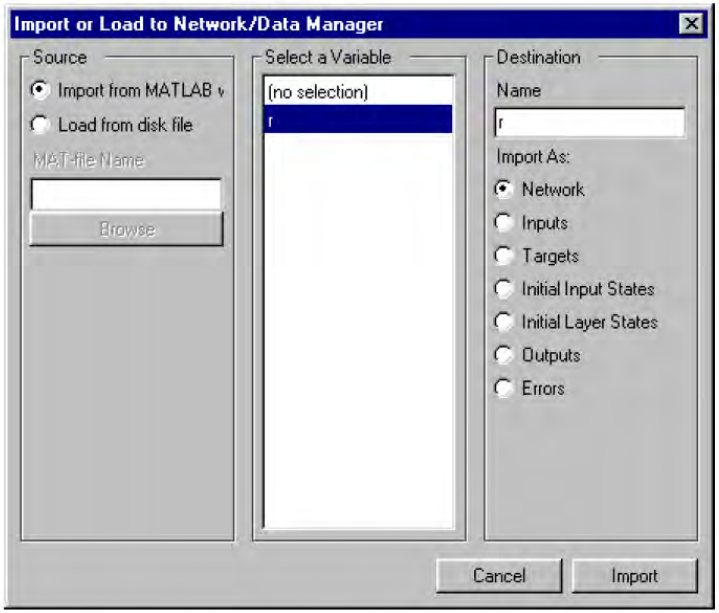
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

Summary

Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem.

The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem.

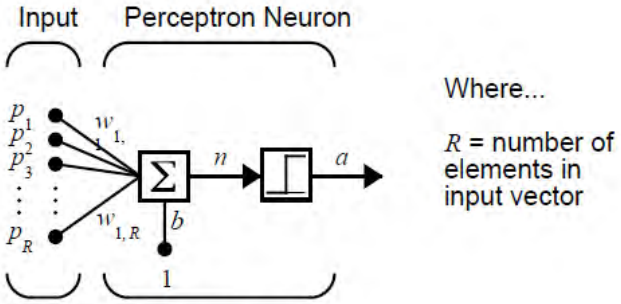
Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution.

Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors.

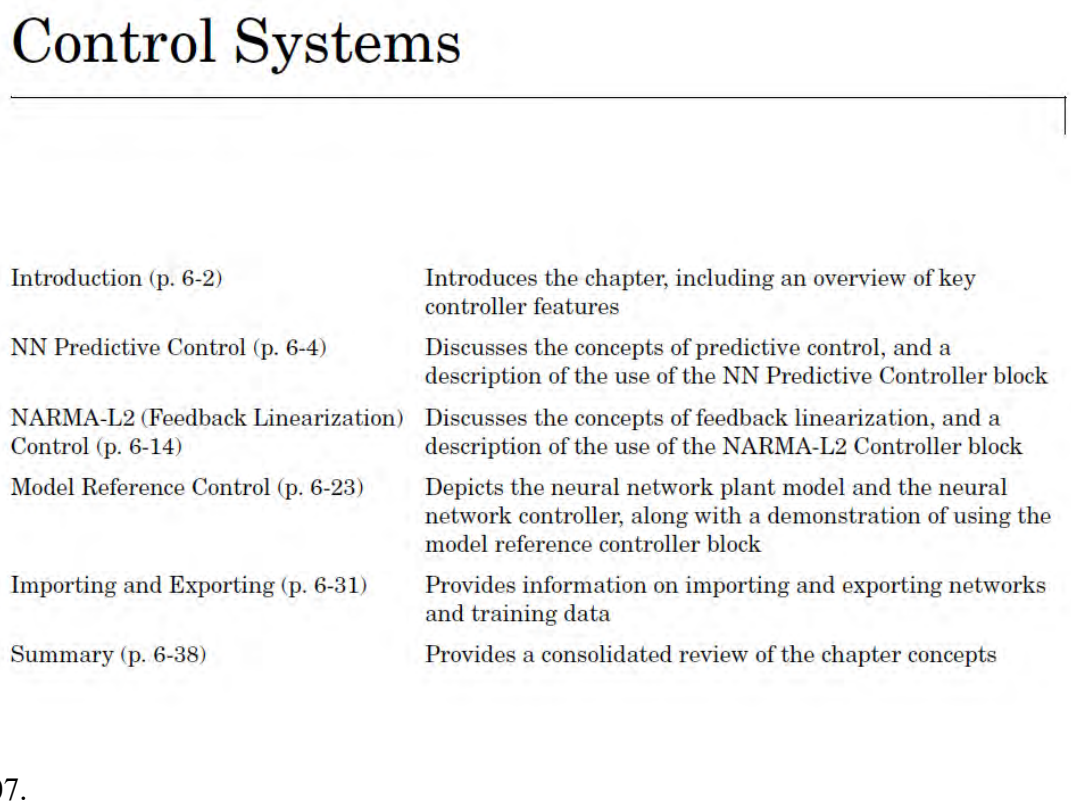
A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace.

Figures and Equations

Perceptron Neuron

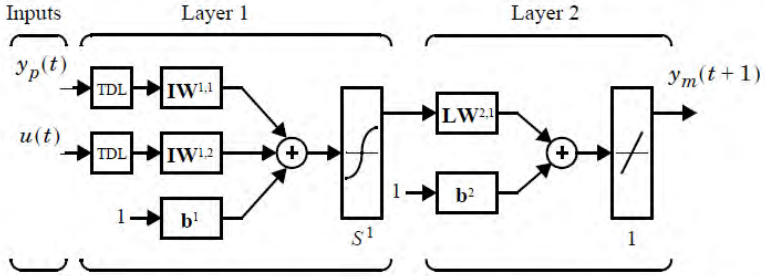


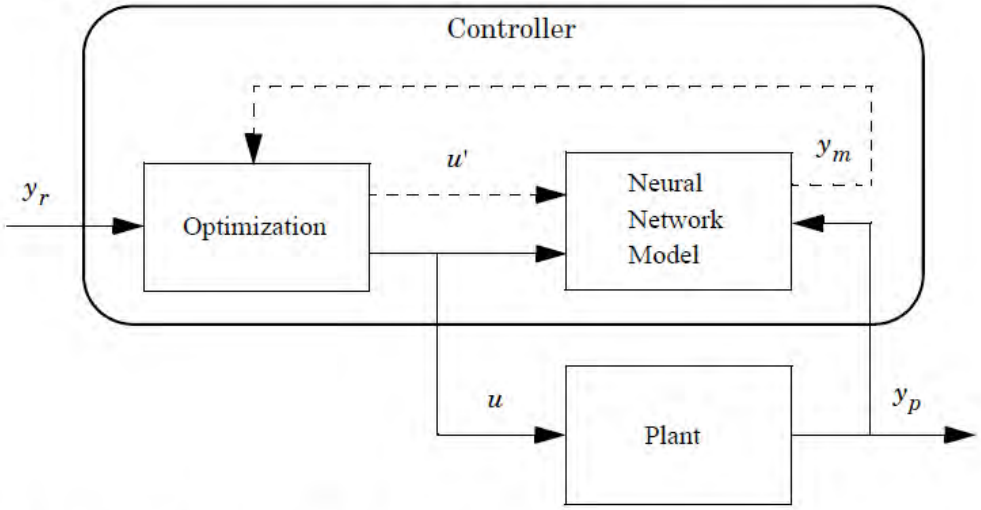
$$a = \mathbf{hardlim}(\mathbf{Wp} + b)$$

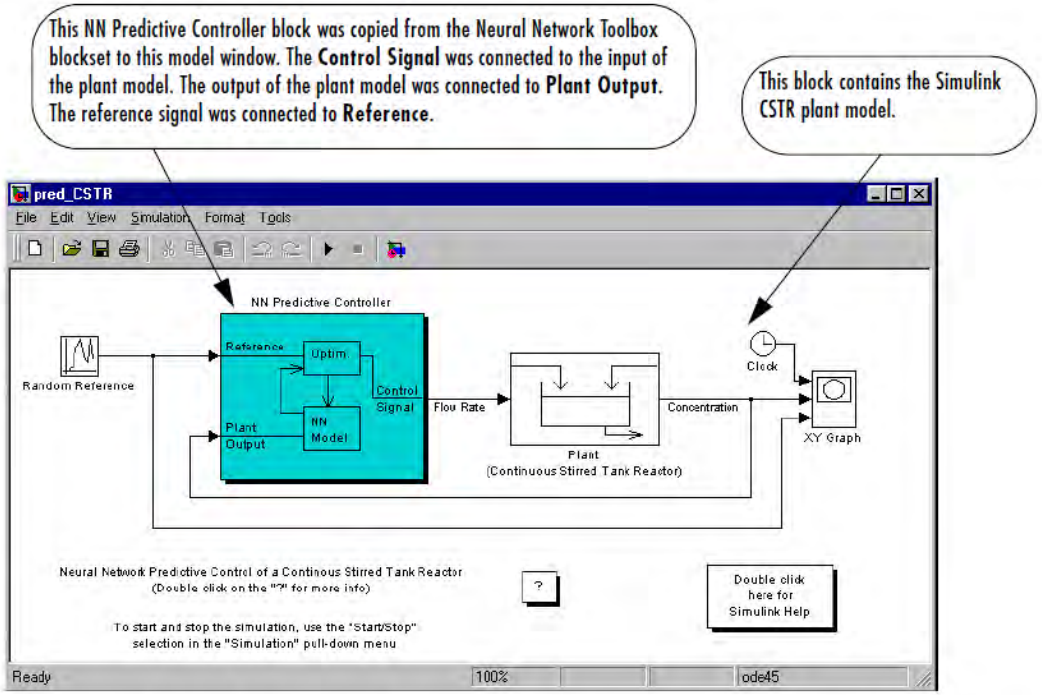
U.S. Patent No. RE49,461	nnet
	101.  Control Systems Introduction (p. 6-2) — Introduces the chapter, including an overview of key controller features NN Predictive Control (p. 6-4) — Discusses the concepts of predictive control, and a description of the use of the NN Predictive Controller block NARMA-L2 (Feedback Linearization) Control (p. 6-14) — Discusses the concepts of feedback linearization, and a description of the use of the NARMA-L2 Controller block Model Reference Control (p. 6-23) — Depicts the neural network plant model and the neural network controller, along with a demonstration of using the model reference controller block Importing and Exporting (p. 6-31) — Provides information on importing and exporting networks and training data Summary (p. 6-38) — Provides a consolidated review of the chapter concepts 207.

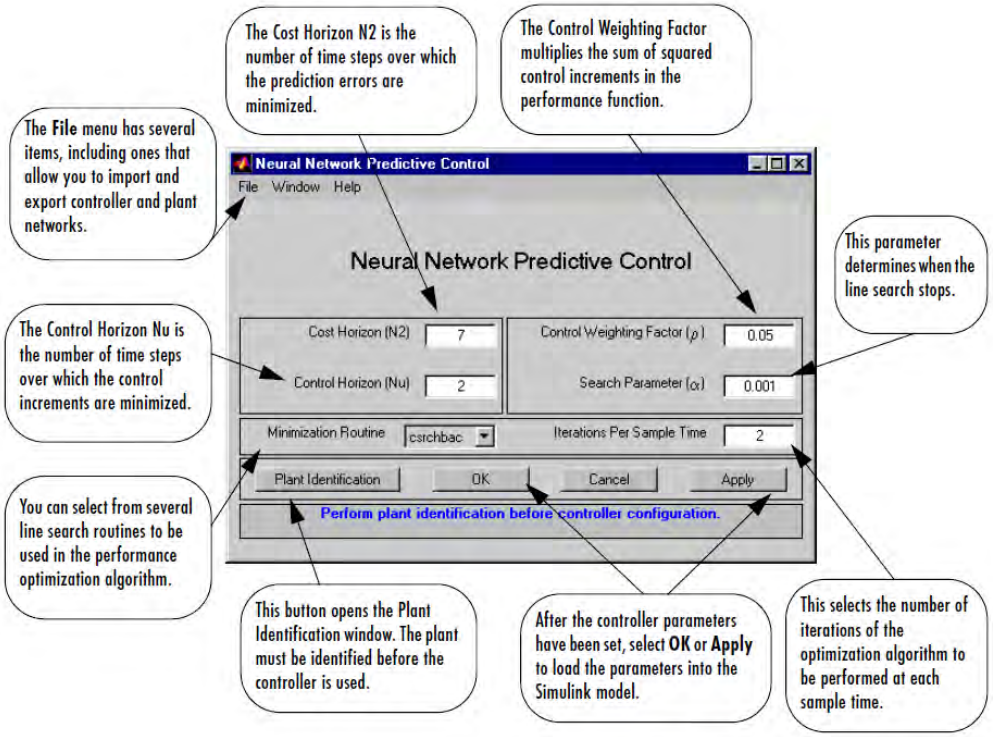
U.S. Patent No. RE49,461	nnet
	Introduction Neural networks have been applied very successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99]. This chapter introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox: • Model Predictive Control • NARMA-L2 (or Feedback Linearization) Control • Model Reference Control This chapter presents brief descriptions of each of these architectures and demonstrates how you can use them. There are typically two steps involved when using neural networks for control: **1** System Identification **2** Control Design In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this chapter, the system identification stage is identical. The control design stage, however, is different for each architecture. • For the model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance. • For the NARMA-L2 control, the controller is simply a rearrangement of the plant model. • For the model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training. The next three sections of this chapter discuss model predictive control, NARMA-L2 control and model reference control. Each section consists of a brief 208.

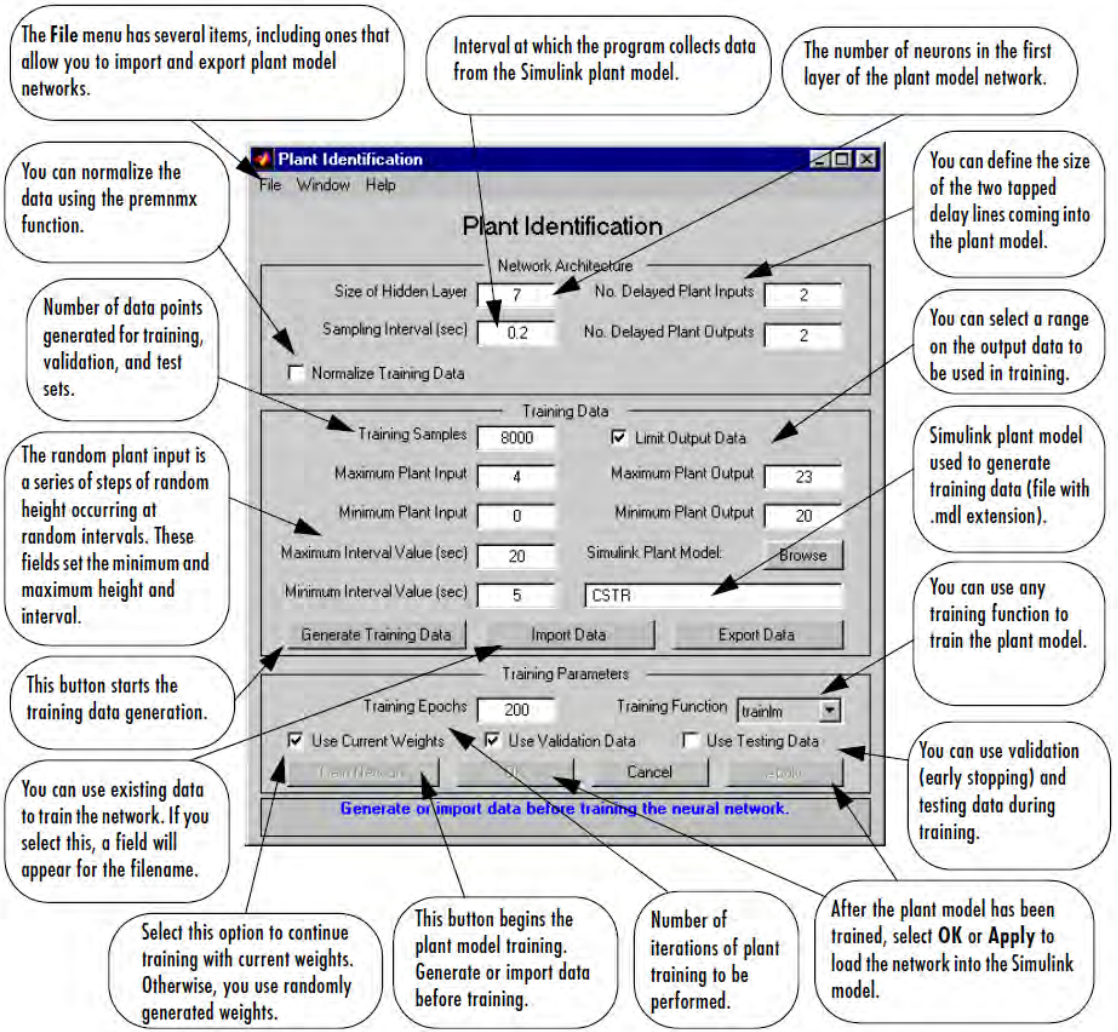
U.S. Patent No. RE49,461	nnet
	description of the control concept, followed by a demonstration of the use of the appropriate Neural Network Toolbox function. These three controllers are implemented as Simulink® blocks, which are contained in the Neural Network Toolbox blockset. To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application. • **Model Predictive Control**. This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form, using any of the training algorithms discussed in Chapter 5. (This is true for all three control architectures.) The controller, however, requires a significant amount of on-line computation, since an optimization algorithm is performed at each sample time to compute the optimal control input. • **NARMA-L2 Control**. This controller requires the least computation of the three architectures described in this chapter. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (The companion form model is described later in this chapter.) • **Model Reference Control**. The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained off-line, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation [HaJe99]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control. 209.

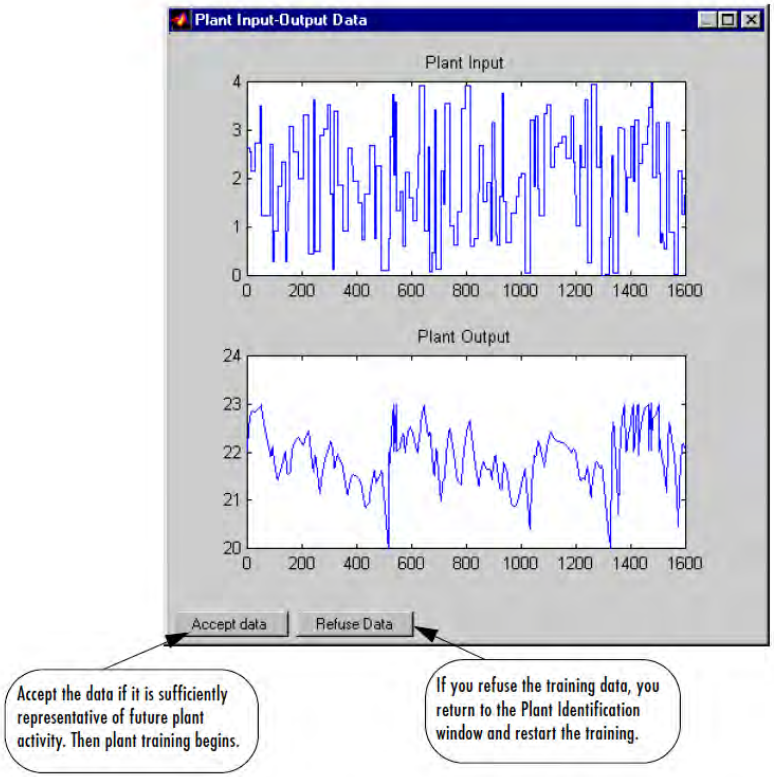
U.S. Patent No. RE49,461	nnet
	The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.  This network can be trained offline in batch mode, using data collected from the operation of the plant. Any of the training algorithms discussed in Chapter 5, "Backpropagation", can be used for network training. This process is discussed in more detail later in this chapter. **Predictive Control** The model predictive control method is based on the receding horizon technique [SoHa96]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon. $$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$ where N_1, N_2 and N_u define the horizons over which the tracking error and the control increments are evaluated. The u' variable is the tentative control signal, y_r is the desired response and y_m is the network model response. The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index. The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J, and 211.

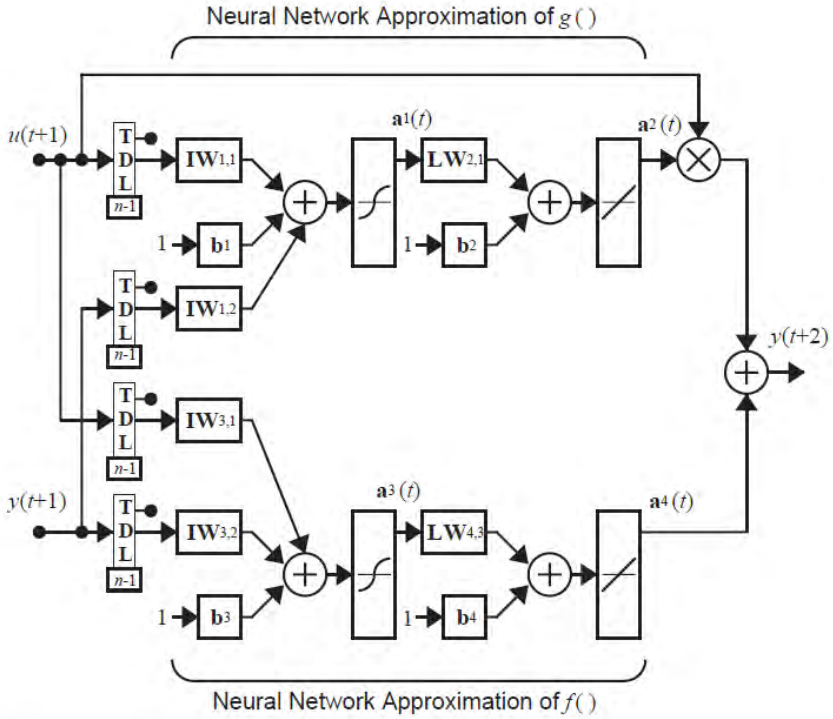
U.S. Patent No. RE49,461	nnet
	then the optimal u is input to the plant. The controller block has been implemented in Simulink, as described in the following section.  **Using the NN Predictive Controller Block** This section demonstrates how the NN Predictive Controller block is used.The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration. A demo model is provided with the Neural Network Toolbox to demonstrate the predictive controller. This demo uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure. 212.

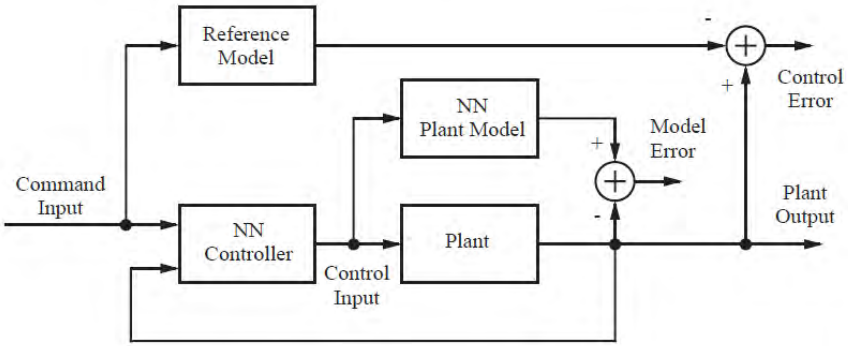
U.S. Patent No. RE49,461	nnet
	 **3** Double-click the NN Predictive Controller block. This brings up the following window for designing the model predictive controller. This window enables you to change the controller horizons N_2 and N_u. (N_1 is fixed at 1.) The weighting parameter ρ, described earlier, is also defined in this window. The parameter α is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in Chapter 5, "Backpropagation." 214.

U.S. Patent No. RE49,461	nnet
	 **4** Select **Plant Identification**. This opens the following window. The neural network plant model must be developed before the controller is used. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. The size of that layer, the number of delayed inputs and delayed outputs, and the training function are selected in this window. You can select any of the training functions described in Chapter 5, "Backpropagation", to train the neural network plant model. 215.

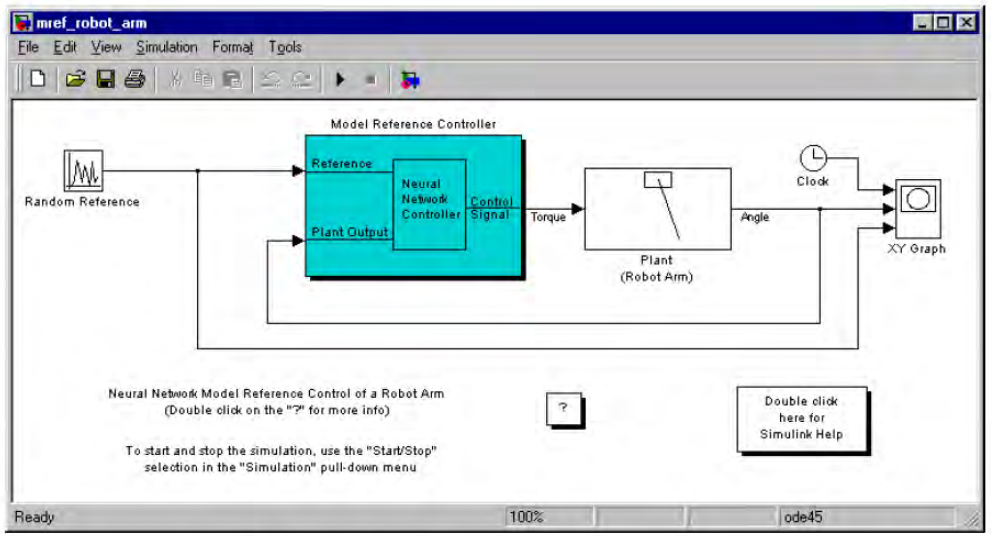
U.S. Patent No. RE49,461	nnet
	 216.

U.S. Patent No. RE49,461	nnet
	5 Select the **Generate Training Data** button. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.  **6** Select **Accept Data**, and then select **Train Network** from the Plant Identification window. Plant model training begins. The training proceeds 217.

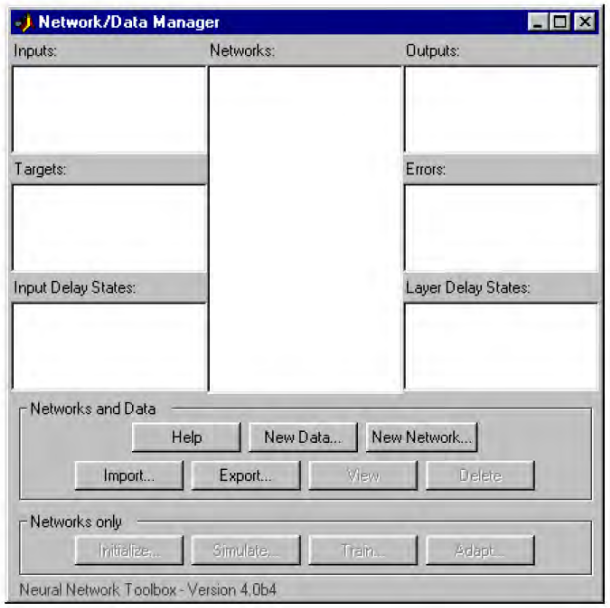
U.S. Patent No. RE49,461	nnet
	 **NARMA-L2 Controller** Using the NARMA-L2 model, you can obtain the controller $$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$ which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller. 222.

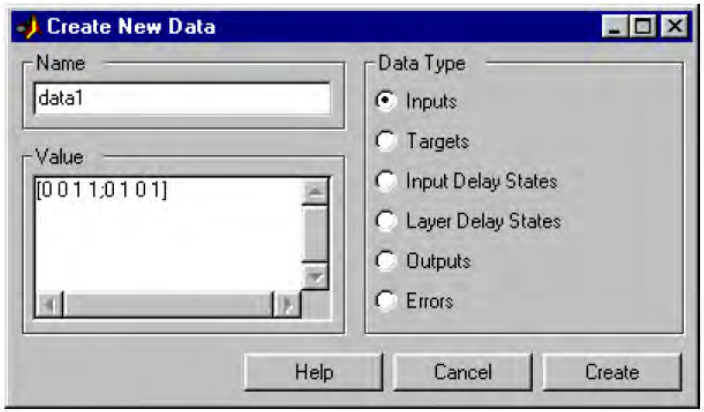
U.S. Patent No. RE49,461	nnet
229.	**Model Reference Control** The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.  The figure on the following page shows the details of the neural network plant model and the neural network controller, as they are implemented in the Neural Network Toolbox. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs: • Delayed reference inputs • Delayed controller outputs • Delayed plant outputs For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model: • Delayed controller outputs • Delayed plant outputs As with the controller, you can set the number of delays. The next section demonstrates how you can set the parameters.

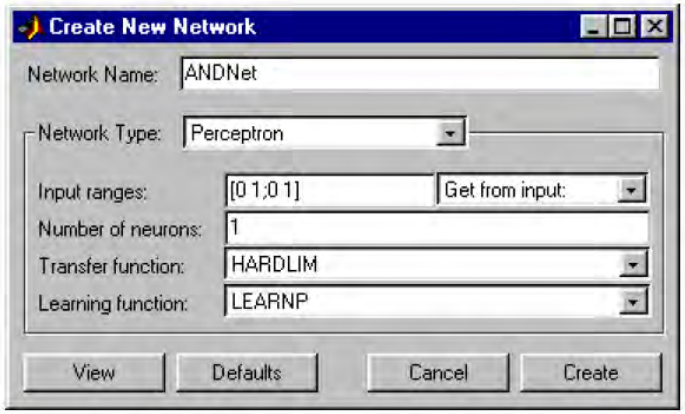
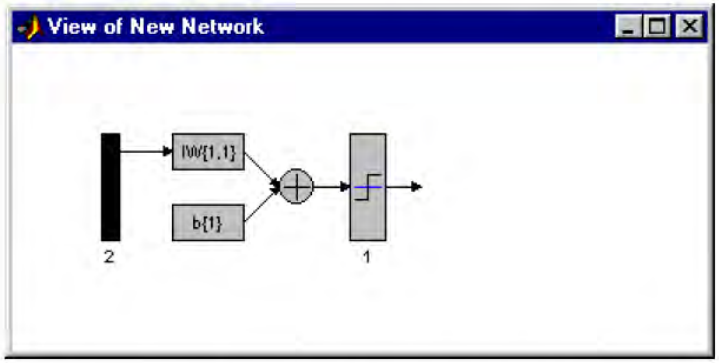
U.S. Patent No. RE49,461	nnet
	Summary The following table summarizes the controllers discussed in this chapter. **Block** / **Description** NN Predictive Control — Uses a neural network plant model to predict future plant behavior. An optimization algorithm determines the control input that optimizes plant performance over a finite time horizon. The plant training requires only a batch algorithm for static networks and is reasonably fast. The controller requires an online optimization algorithm, which requires more computation than the other controllers. NARMA-L2 Control — An approximate plant model is in companion form. The next control input is computed to force the plant output to follow a reference signal. The neural network plant model is trained with static backpropagation and is reasonably fast. The controller is a rearrangement of the plant model, and requires minimal online computation. Model Reference Control — A neural network plant model is first developed. The plant model is then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a more general class of plant than does the NARMA-L2 control architecture. The controller requires minimal online computation. 244.

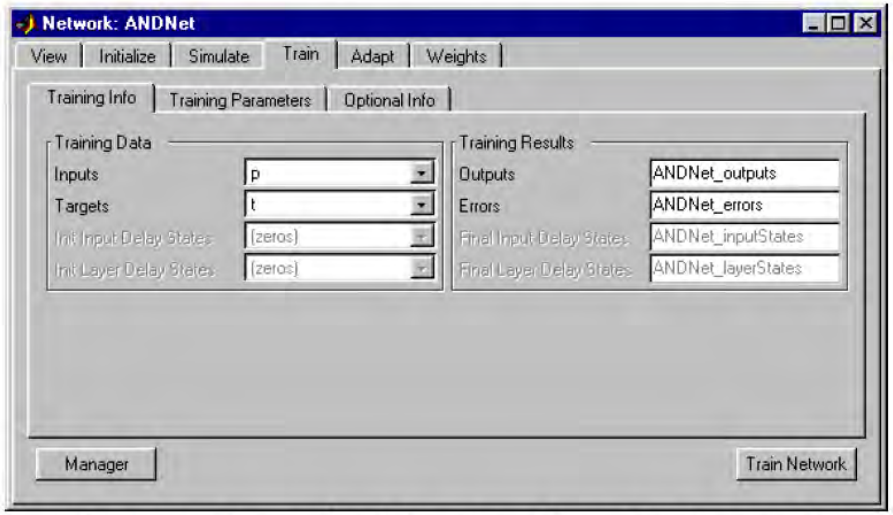
U.S. Patent No. RE49,461	nnet
	This demo uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used. To run this demo, follow these steps. **1** Start MATLAB. **2** Run the demo model by typing `mrefrobotarm` in the MATLAB command window. This command starts Simulink and creates the following model window. The Model Reference Control block has already been placed in the model.  **3** Double-click the Model Reference Control block. This brings up the following window for training the model reference controller. 232.

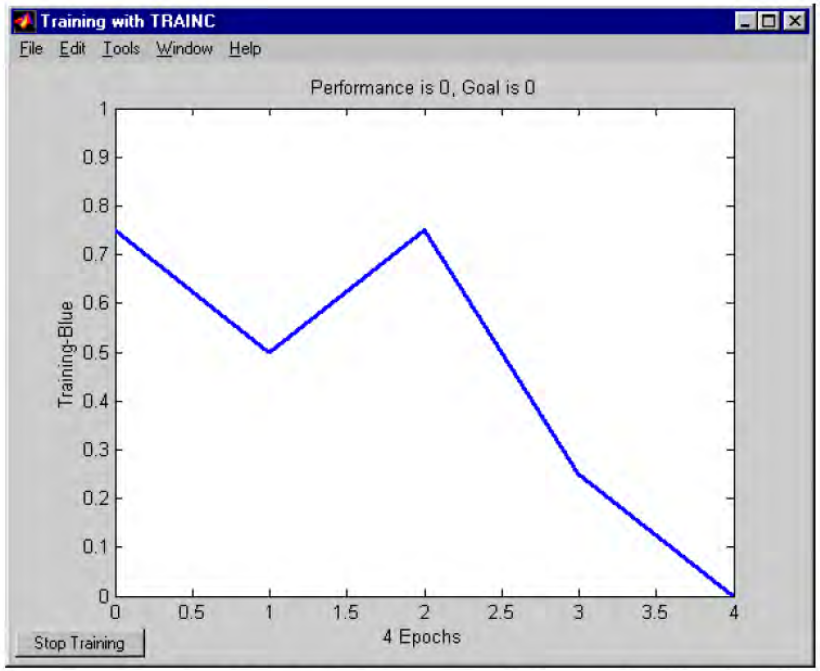
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[c] shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network;	nnet discloses, expressly and/or inherently, shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network. *See* Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting the execution of the computations representing the artificial neural network (e.g., GUI, nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name `t`, specify the value `[0 0 0 1]`, and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter `ANDNet` under **Network Name**. Set the **Network Type** to `Perceptron`, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges `[0 1;0 1]`.We want to use a `hardlim` transfer function and a `learnp` learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
   2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
   ans =
   -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

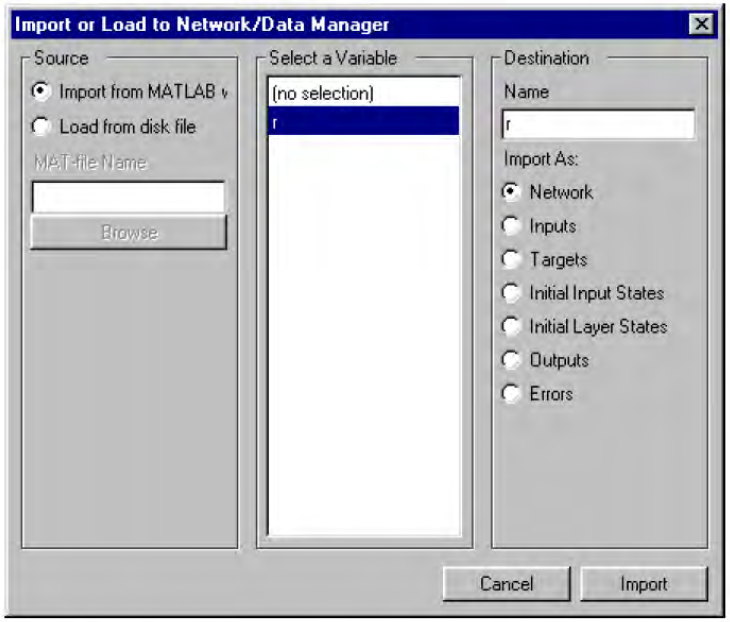
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

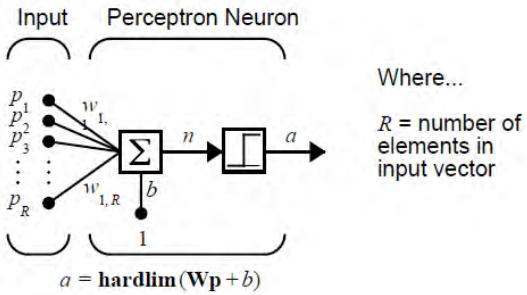
Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

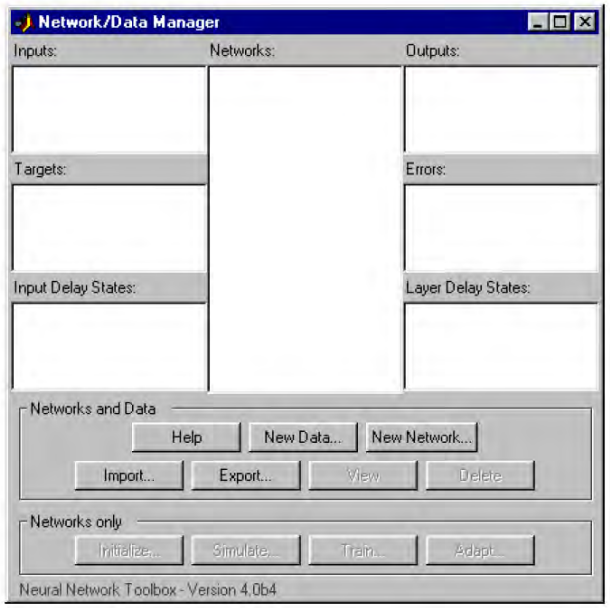
U.S. Patent No. RE49,461	nnet

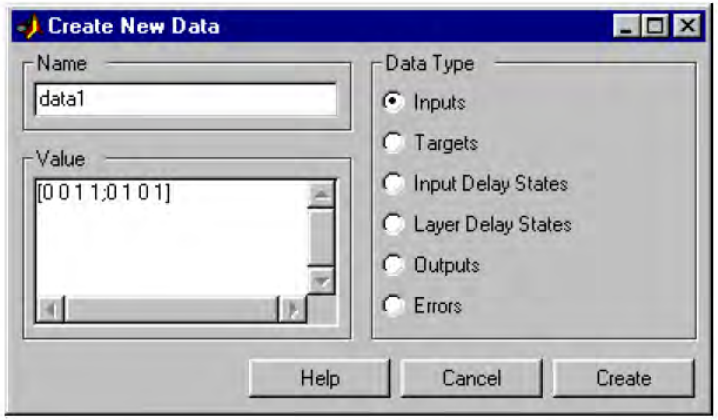
	Simulation With Concurrent Inputs in a Dynamic Network
	If we were to apply the same inputs from the previous example as a set of concurrent inputs instead of a sequence of inputs, we would obtain a completely different response. (Although, it is not clear why we would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, if we use a concurrent set of inputs we have
	$\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix}3\end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix}4\end{bmatrix}$
	which can be created with the following code:
	`P = [1 2 3 4];`
	When we simulate with concurrent inputs we obtain
	`A = sim(net,P)`
	`A =`
	`1 2 3 4`
	The result is the same as if we had concurrently applied each one of the inputs to a separate network and computed one output. Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero. For this case the output will simply be 1 times the input, since the weight that multiplies the current input is 1.
	In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences. For example, let's say we wanted to present the following two sequences to the network:
	$\mathbf{p}_1(1) = \begin{bmatrix}1\end{bmatrix}, \mathbf{p}_1(2) = \begin{bmatrix}2\end{bmatrix}, \mathbf{p}_1(3) = \begin{bmatrix}3\end{bmatrix}, \mathbf{p}_1(4) = \begin{bmatrix}4\end{bmatrix}$
	$\mathbf{p}_2(1) = \begin{bmatrix}4\end{bmatrix}, \mathbf{p}_2(2) = \begin{bmatrix}3\end{bmatrix}, \mathbf{p}_2(3) = \begin{bmatrix}2\end{bmatrix}, \mathbf{p}_2(4) = \begin{bmatrix}1\end{bmatrix}$
	52.

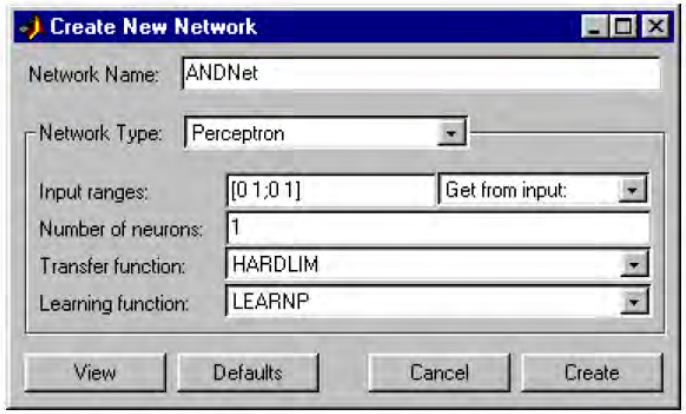
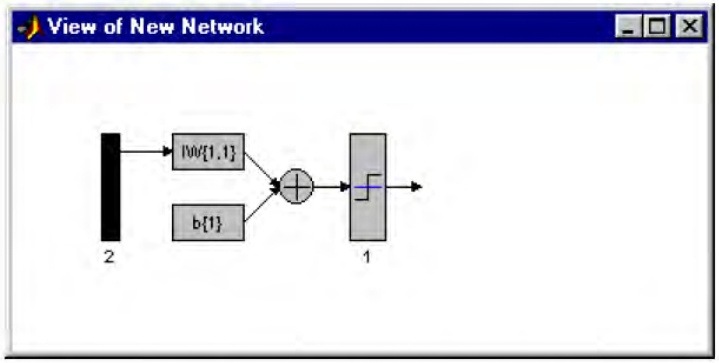
U.S. Patent No. RE49,461	nnet

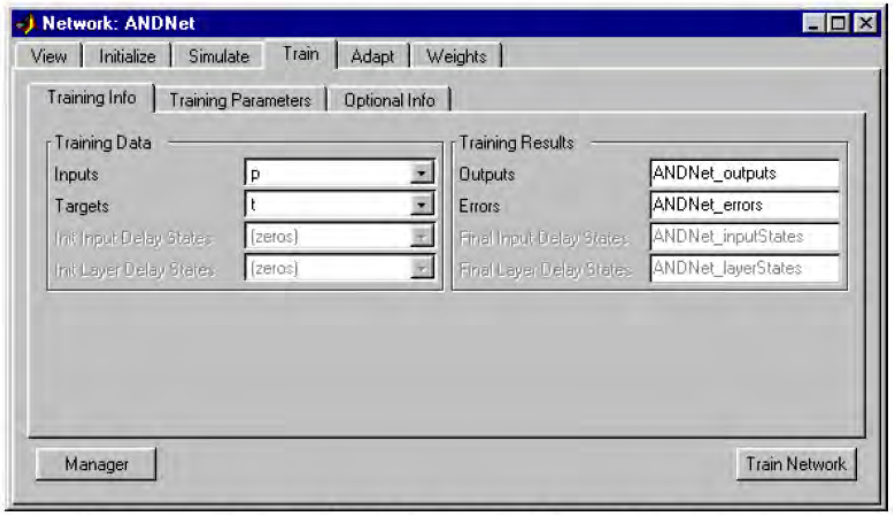
53.	The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:
	`P = {[1 4] [2 3] [3 2] [4 1]};`
	We can now simulate the network:
	`A = sim(net,P);`
	The resulting network output would be
	`A = {[ 1 4] [4 11] [7 8] [10 5]}`
	As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.
	The following diagram shows the general format for the input `P` to the `sim` function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.
	Qth Sequence
	$\{[\mathbf{p}_1(1), \mathbf{p}_2(1), \ldots, \mathbf{p}_Q(1)], [\mathbf{p}_1(2), \mathbf{p}_2(2), \ldots, \mathbf{p}_Q(2)], \ldots, [\mathbf{p}_1(TS), \mathbf{p}_2(TS), \ldots, \mathbf{p}_Q(TS)]\}$
	First Sequence
	In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.

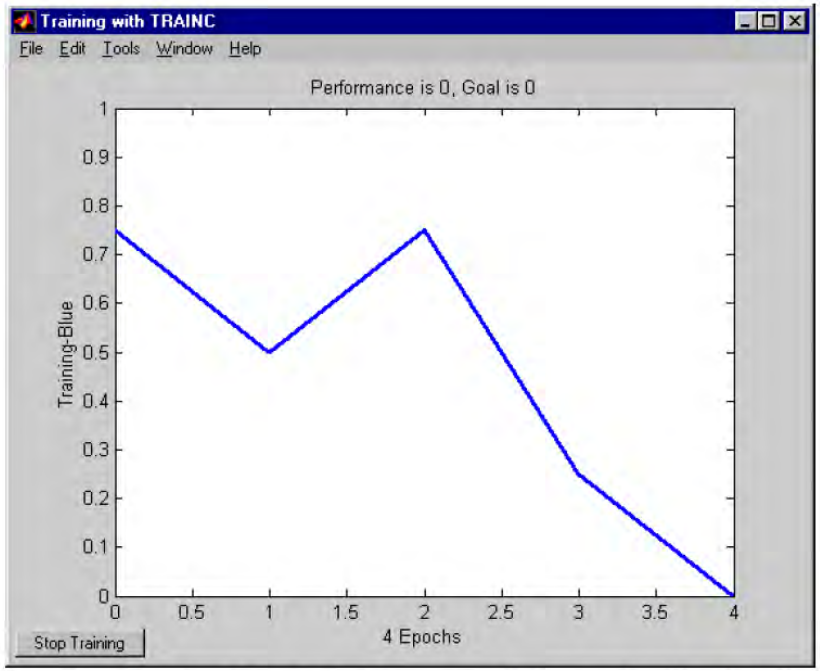
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[d] shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network;	nnet discloses, expressly and/or inherently, shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network. *See* Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network (e.g., GUI, nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name `t`, specify the value `[0 0 0 1]`, and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter `ANDNet` under **Network Name**. Set the **Network Type** to `Perceptron`, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges `[0 1;0 1]`.We want to use a `hardlim` transfer function and a `learnp` learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

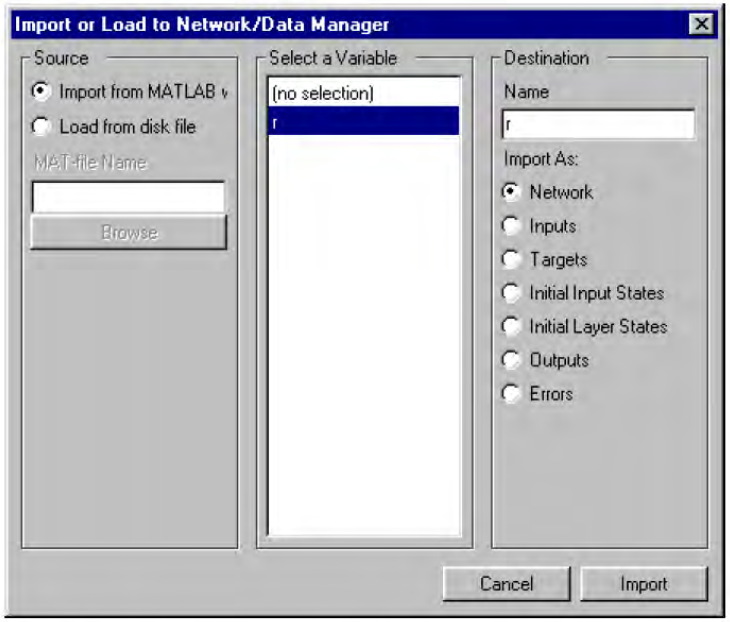
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

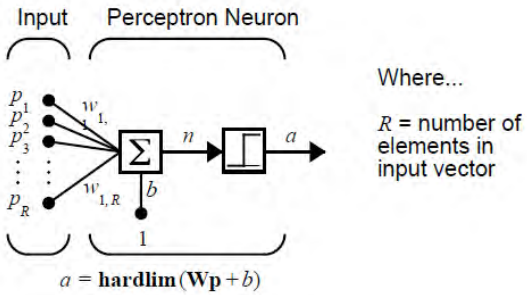
U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

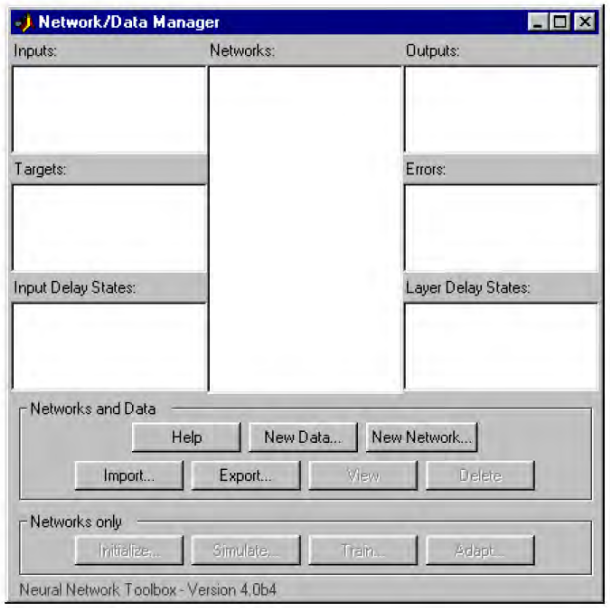
U.S. Patent No. RE49,461	nnet

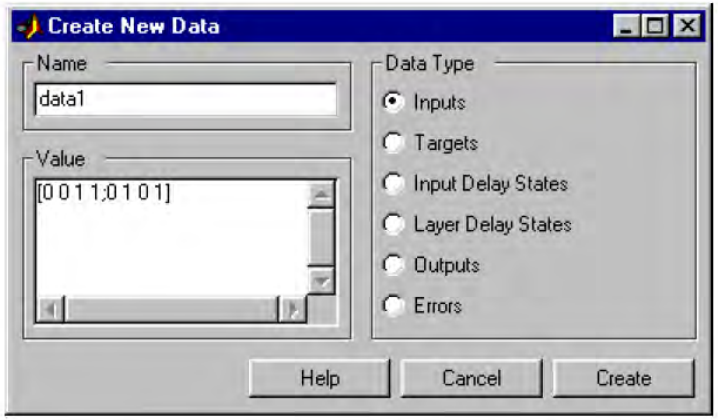
	Simulation With Concurrent Inputs in a Dynamic Network
	If we were to apply the same inputs from the previous example as a set of concurrent inputs instead of a sequence of inputs, we would obtain a completely different response. (Although, it is not clear why we would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, if we use a concurrent set of inputs we have
	$\mathbf{p}_1 = [1], \quad \mathbf{p}_2 = [2], \quad \mathbf{p}_3 = [3], \quad \mathbf{p}_4 = [4]$
	which can be created with the following code:
	`P = [1 2 3 4];`
	When we simulate with concurrent inputs we obtain
	`A = sim(net,P)`
	`A =`
	`1 2 3 4`
	The result is the same as if we had concurrently applied each one of the inputs to a separate network and computed one output. Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero. For this case the output will simply be 1 times the input, since the weight that multiplies the current input is 1.
	In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences. For example, let's say we wanted to present the following two sequences to the network:
	$\mathbf{p}_1(1) = [1], \mathbf{p}_1(2) = [2], \mathbf{p}_1(3) = [3], \mathbf{p}_1(4) = [4]$
	$\mathbf{p}_2(1) = [4], \mathbf{p}_2(2) = [3], \mathbf{p}_2(3) = [2], \mathbf{p}_2(4) = [1]$
	52.

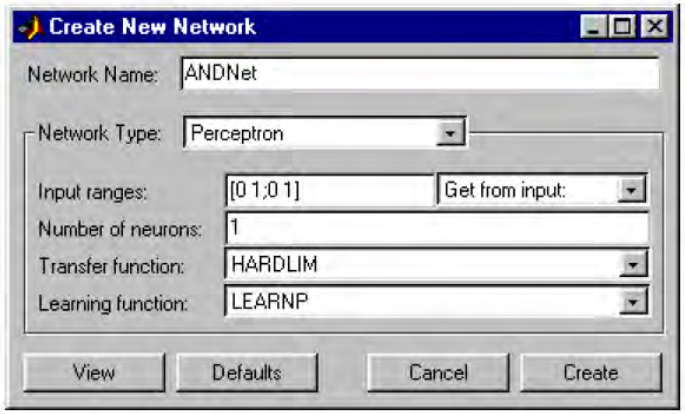
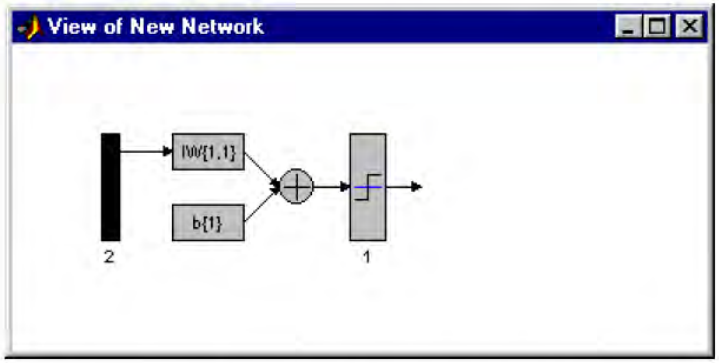
U.S. Patent No. RE49,461	nnet

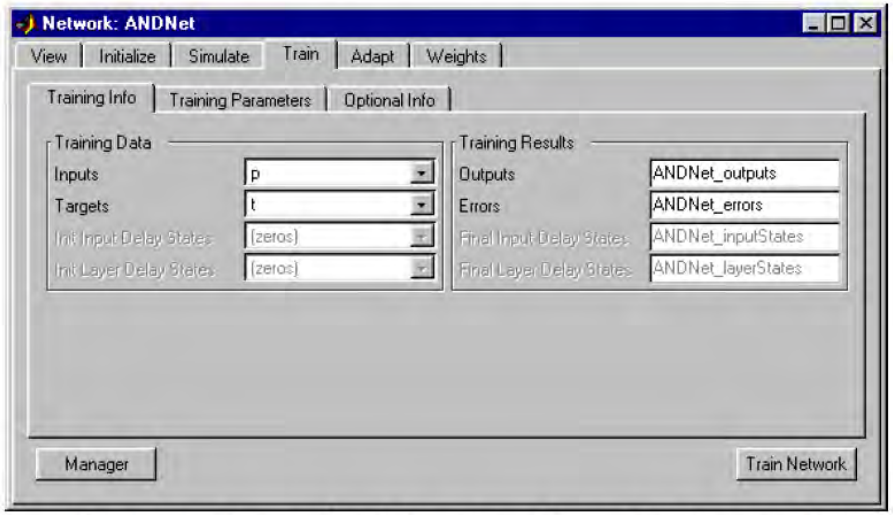
53.	The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:
	`P = {[1 4] [2 3] [3 2] [4 1]};`
	We can now simulate the network:
	`A = sim(net,P);`
	The resulting network output would be
	`A = {[ 1 4] [4 11] [7 8] [10 5]}`
	As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.
	The following diagram shows the general format for the input `P` to the `sim` function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.
	Qth Sequence
	$\{[\mathbf{p}_1(1), \mathbf{p}_2(1), \ldots, \mathbf{p}_Q(1)], [\mathbf{p}_1(2), \mathbf{p}_2(2), \ldots, \mathbf{p}_Q(2)], \ldots, [\mathbf{p}_1(TS), \mathbf{p}_2(TS), \ldots, \mathbf{p}_Q(TS)]\}$
	First Sequence
	In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.

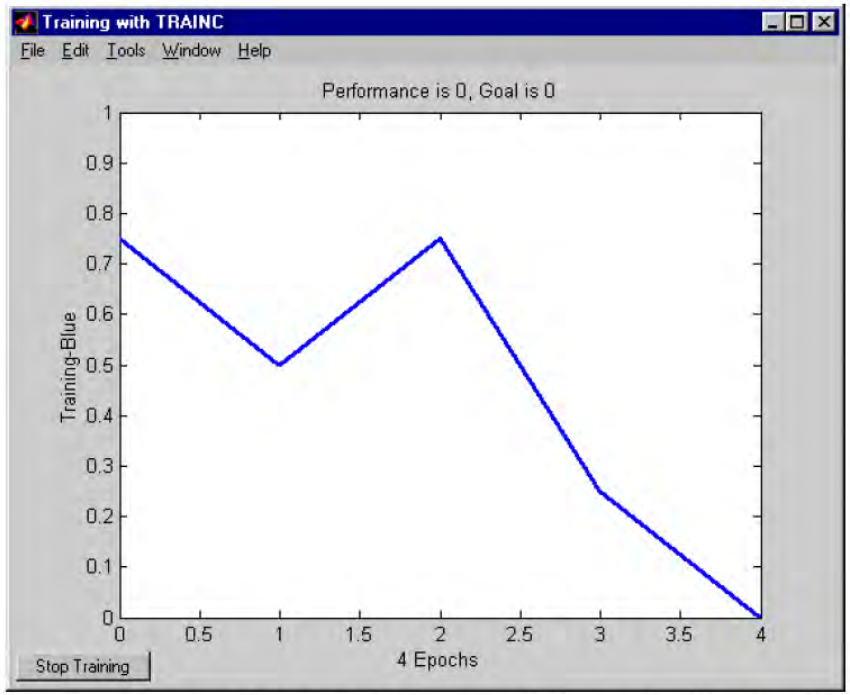
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[e] queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network; and	nnet discloses, expressly and/or inherently, queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network. *See* Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses queueing a user command received by the user interaction stream during execution of computations representing the artificial neural network (e.g., GUI, nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
   2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
   ans =
   -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

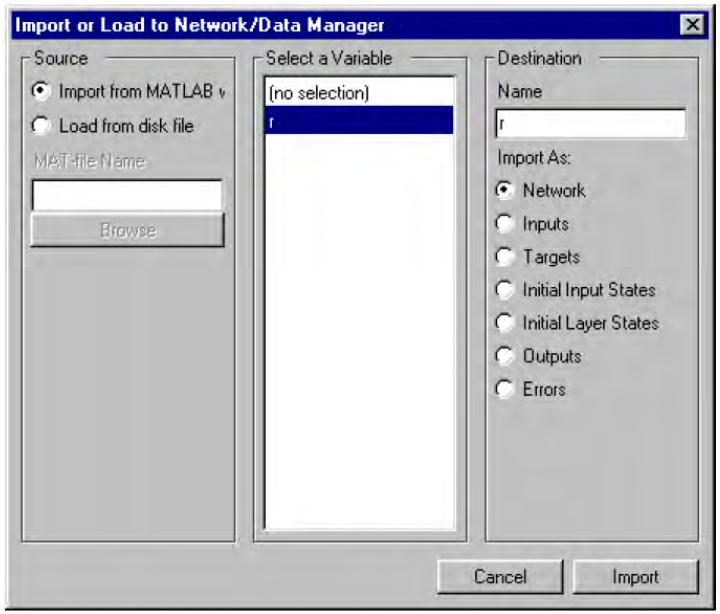
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

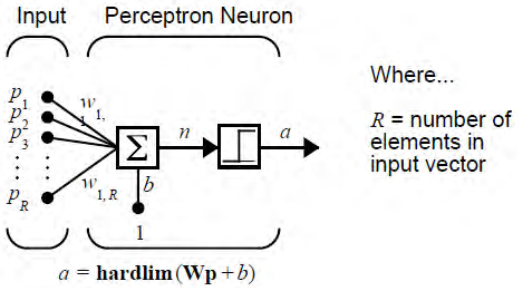
Create a new vector.

```
   r= [0; 1; 2; 3]
   r =
        0
        1
        2
        3
```

Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

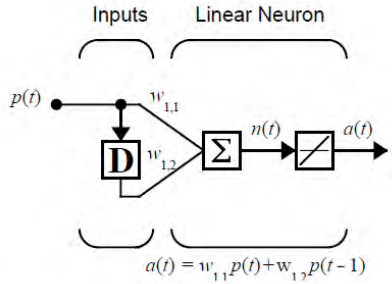
U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
101.	**Summary** Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  $a = \mathbf{hardlim}(\mathbf{Wp} + b)$

U.S. Patent No. RE49,461	nnet

	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \end{bmatrix}$.
	The commands for these assignments are
	`net.IW{1,1} = [1 2];`
	`net.b{1} = 0;`
	Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:
	$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix},$
	Concurrent vectors are presented to the network as a single matrix:
	`P = [1 2 2 3; 2 1 3 1];`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`5 4 8 5`
	A single matrix of concurrent vectors is presented to the network and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important as they do not interact with each other.
	Simulation With Sequential Inputs in a Dynamic Network
	When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, we use a simple network that contains one delay.
	50.

U.S. Patent No. RE49,461	nnet

	
	The following commands create this network:
	`net = newlin([-1 1],1,[0 1]);`
	`net.biasConnect = 0;`
	Assign the weight matrix to be
	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$.
	The command is
	`net.IW{1,1} = [1 2];`
	Suppose that the input sequence is
	$\mathbf{p}1 = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{p}2 = \begin{bmatrix} 2 \end{bmatrix}$, $\mathbf{p}3 = \begin{bmatrix} 3 \end{bmatrix}$, $\mathbf{p}4 = \begin{bmatrix} 4 \end{bmatrix}$,
	Sequential inputs are presented to the network as elements of a cell array:
	`P = {1 2 3 4};`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`[1] [4] [7] [10]`
	We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case,
	
	51.
	
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the

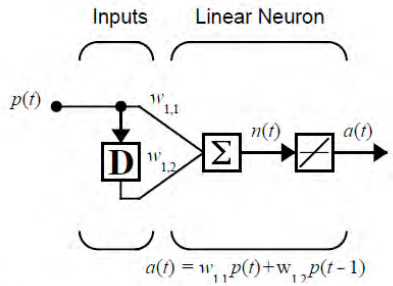
U.S. Patent No. RE49,461	nnet
	teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
21[f] executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream.	nnet discloses, expressly and/or inherently, executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream. *See* Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream (e.g., GUI).*

U.S. Patent No. RE49,461	nnet
91.	# Graphical User Interface ## Introduction to the GUI The graphical user interface (GUI) is designed to be simple and user friendly, but we will go through a simple example to get you started. In what follows you bring up a GUI **Network/Data Manager** window. This window has its own work area, separate from the more familiar command line workspace. Thus, when using the GUI, you might "export" the GUI results to the (command line) workspace. Similarly you may want to "import" results from the command line workspace to the GUI. Once the **Network/Data Manager** is up and running, you can create a network, view it, train it, simulate it and export the final results to the workspace. Similarly, you can import data from the workspace for use in the GUI. The following example deals with a perceptron network. We go through all the steps of creating a network and show you what you might expect to see as you go along. ## Create a Perceptron Network (nntool) We create a perceptron network to perform the AND function in this example. It has an input vector `p= [0 0 1 1;0 1 0 1]` and a target vector `t=[0 0 0 1]`. We call the network `ANDNet`. Once created, the network will be trained. We can then save the network, its output, etc., by "exporting" it to the command line. ### Input and target To start, type `nntool`. The following window appears.

U.S. Patent No. RE49,461	nnet

	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \end{bmatrix}$.
	The commands for these assignments are
	`net.IW{1,1} = [1 2];`
	`net.b{1} = 0;`
	Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:
	$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix},$
	Concurrent vectors are presented to the network as a single matrix:
	`P = [1 2 2 3; 2 1 3 1];`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`5 4 8 5`
	A single matrix of concurrent vectors is presented to the network and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important as they do not interact with each other.
	Simulation With Sequential Inputs in a Dynamic Network
	When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, we use a simple network that contains one delay.
	50.

U.S. Patent No. RE49,461	nnet

51.	
	The following commands create this network:
	`net = newlin([-1 1],1,[0 1]);`
	`net.biasConnect = 0;`
	Assign the weight matrix to be
	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$.
	The command is
	`net.IW{1,1} = [1 2];`
	Suppose that the input sequence is
	$\mathbf{p}1 = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{p}2 = \begin{bmatrix} 2 \end{bmatrix}$, $\mathbf{p}3 = \begin{bmatrix} 3 \end{bmatrix}$, $\mathbf{p}4 = \begin{bmatrix} 4 \end{bmatrix}$,
	Sequential inputs are presented to the network as elements of a cell array:
	`P = {1 2 3 4};`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`[1] [4] [7] [10]`
	We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case,

U.S. Patent No. RE49,461	nnet

	The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:
	`P = {[1 4] [2 3] [3 2] [4 1]};`
	We can now simulate the network:
	`A = sim(net,P);`
	The resulting network output would be
	`A = {[ 1 4] [4 11] [7 8] [10 5]}`
	As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.
	The following diagram shows the general format for the input `P` to the `sim` function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.
	Qth Sequence
	$\{[\mathbf{p}_1(1), \mathbf{p}_2(1), \ldots, \mathbf{p}_Q(1)], [\mathbf{p}_1(2), \mathbf{p}_2(2), \ldots, \mathbf{p}_Q(2)], \ldots, [\mathbf{p}_1(TS), \mathbf{p}_2(TS), \ldots, \mathbf{p}_Q(TS)]\}$
	First Sequence
	In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.
	53.

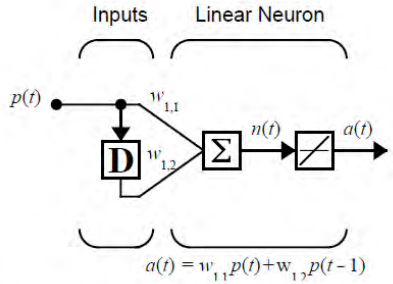
U.S. Patent No. RE49,461	nnet
	nnet contemplates temporal neural network behavior. **Summary** The inputs to a neuron include its bias and the sum of its weighted inputs (using the inner product). The output of a neuron depends on the neuron's inputs and on its transfer function. There are many useful transfer functions. A single neuron cannot do very much. However, several neurons can be combined into a layer or multiple layers that have great power. Hopefully this toolbox makes it easy to create and understand such large networks. The architecture of a network consists of a description of how many layers a network has, the number of neurons in each layer, each layer's transfer function, and how the layers connect to each other. The best architecture to use depends on the type of problem to be represented by the network. A network effects a computation by mapping input values to output values. The particular mapping problem to be performed fixes the number of inputs, as well as the number of outputs for the network. Aside from the number of neurons in a network's output layer, the number of neurons in each layer is up to the designer. Except for purely linear networks, the more neurons in a hidden layer, the more powerful the network. If a linear mapping needs to be represented linear neurons should be used. However, linear networks cannot perform any nonlinear computation. Use of a nonlinear transfer function makes a network capable of storing nonlinear relationships between input and output. A very simple problem can be represented by a single layer of neurons. However, single-layer networks cannot solve certain problems. Multiple feed-forward layers give a network greater freedom. For example, any reasonable function can be represented with a two-layer network: a sigmoid layer feeding a linear output layer. Networks with biases can represent relationships between inputs and outputs more easily than networks without biases. (For example, a neuron without a bias will always have a net input to the transfer function of zero when all of its inputs are zero. However, a neuron with a bias can learn to have any net transfer function input under the same conditions by learning an appropriate value for the bias.) Feed-forward networks cannot perform temporal computation. More complex networks with internal feedback paths are required for temporal behavior. 60. To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the

U.S. Patent No. RE49,461	nnet
	teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
22. The method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network.	nnet discloses, expressly and/or inherently, the method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network. *See* claim 21; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses the user interaction stream controlling the data exchange outside of the execution of computations (e.g., nntool, custom functions).*

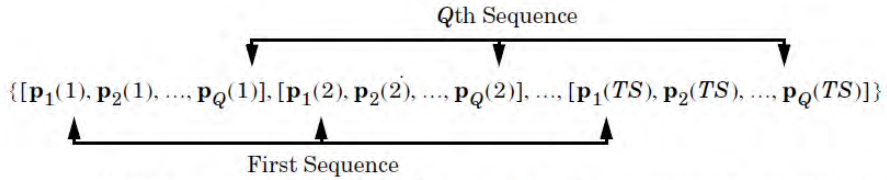
U.S. Patent No. RE49,461	nnet

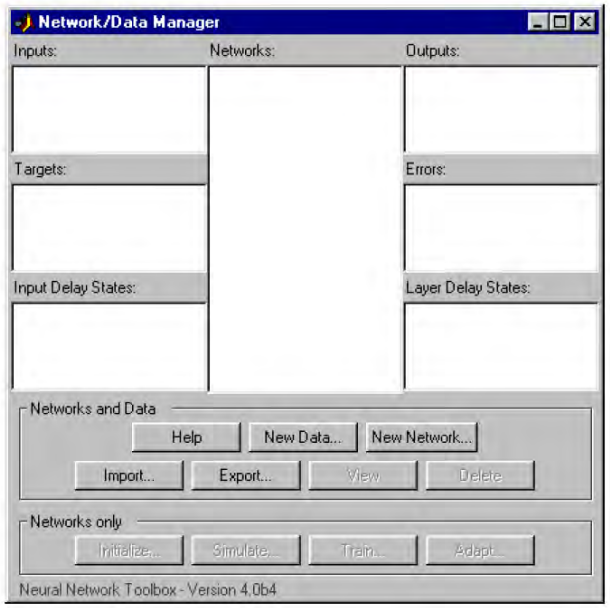
	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \end{bmatrix}$.
	The commands for these assignments are
	`net.IW{1,1} = [1 2];`
	`net.b{1} = 0;`
	Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:
	$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix},$
	Concurrent vectors are presented to the network as a single matrix:
	`P = [1 2 2 3; 2 1 3 1];`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`5 4 8 5`
	A single matrix of concurrent vectors is presented to the network and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important as they do not interact with each other.
	Simulation With Sequential Inputs in a Dynamic Network
	When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, we use a simple network that contains one delay.
	50.

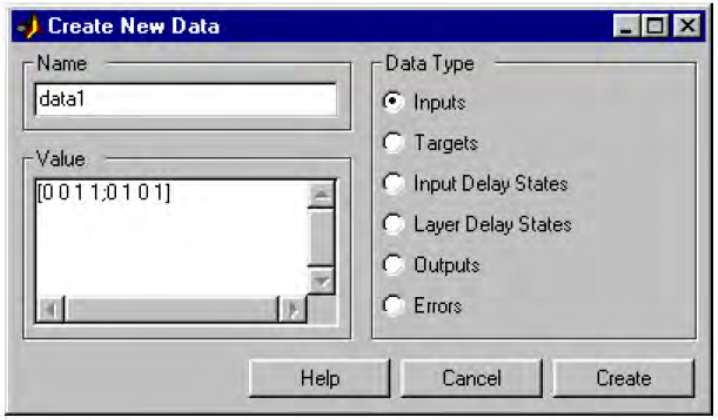
U.S. Patent No. RE49,461	nnet

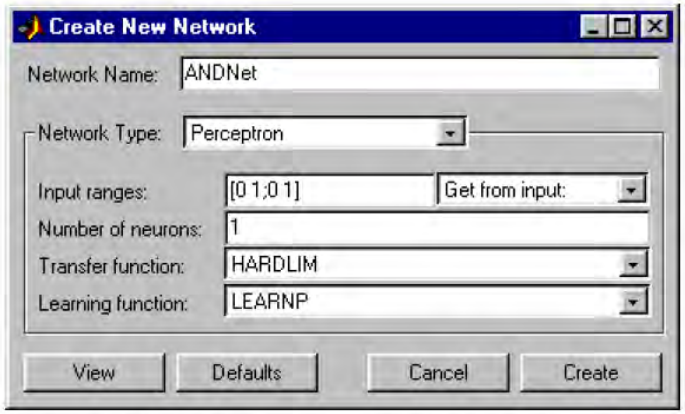
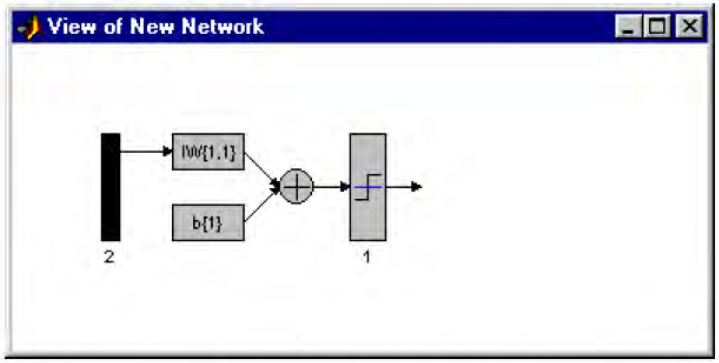
51.	
	The following commands create this network:
	`net = newlin([-1 1],1,[0 1]);`
	`net.biasConnect = 0;`
	Assign the weight matrix to be
	$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$.
	The command is
	`net.IW{1,1} = [1 2];`
	Suppose that the input sequence is
	$\mathbf{p}1 = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{p}2 = \begin{bmatrix} 2 \end{bmatrix}$, $\mathbf{p}3 = \begin{bmatrix} 3 \end{bmatrix}$, $\mathbf{p}4 = \begin{bmatrix} 4 \end{bmatrix}$,
	Sequential inputs are presented to the network as elements of a cell array:
	`P = {1 2 3 4};`
	We can now simulate the network:
	`A = sim(net,P)`
	`A =`
	`[1] [4] [7] [10]`
	We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case,

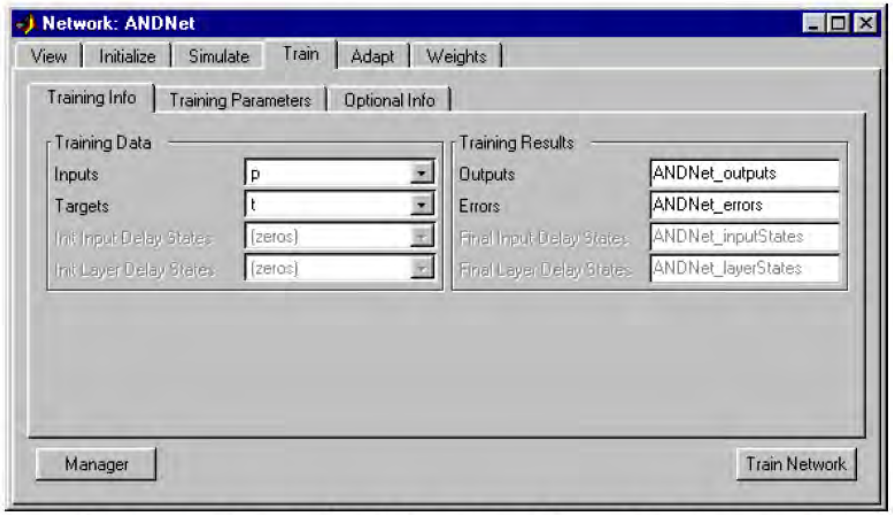
U.S. Patent No. RE49,461	nnet

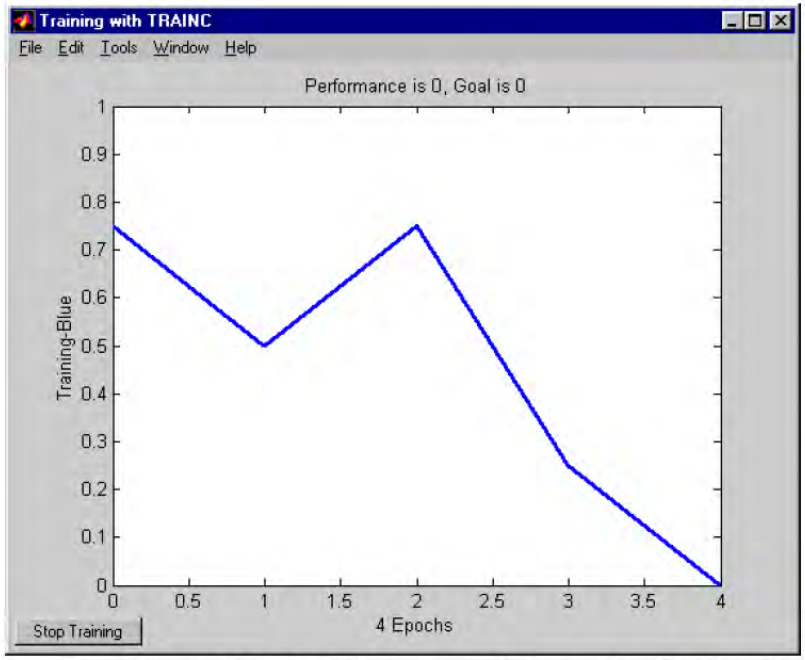
53.	The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:
	`P = {[1 4] [2 3] [3 2] [4 1]};`
	We can now simulate the network:
	`A = sim(net,P);`
	The resulting network output would be
	`A = {[ 1 4] [4 11] [7 8] [10 5]}`
	As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.
	The following diagram shows the general format for the input `P` to the `sim` function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.
	 
	
	In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

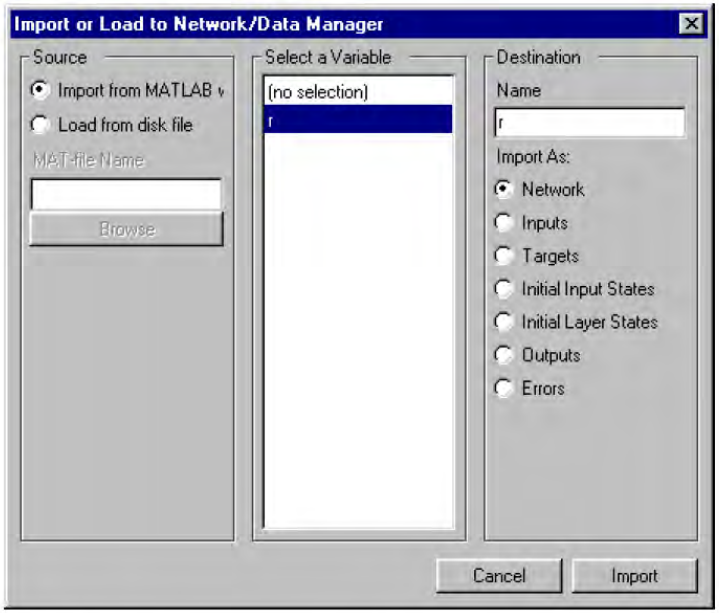
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

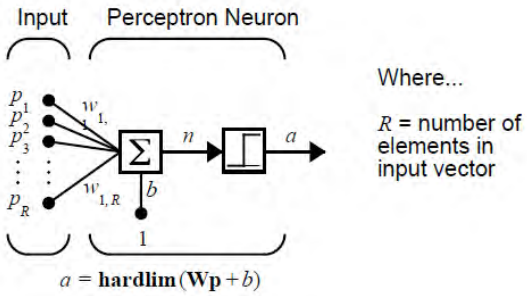
U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  $a = \mathbf{hardlim}(\mathbf{Wp} + b)$ 101.

U.S. Patent No. RE49,461	nnet
	392.

12 Advanced Topics

Custom Functions

The toolbox allows you to create and use many kinds of functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for your networks.

The following sections describe how to create your own versions of these kinds of functions:

- Simulation functions
  - transfer functions
  - net input functions
  - weight functions
- Initialization functions
  - network initialization functions
  - layer initialization functions
  - weight and bias initialization functions
- Learning functions
  - network training functions
  - network adapt functions
  - network performance functions
  - weight and bias learning functions
- Self-organizing map functions
  - topology functions
  - distance functions

Simulation Functions

You can create three kinds of simulation functions: transfer, net input, and weight functions. You can also provide associated derivative functions to enable backpropagation learning with your functions.

Transfer Functions

Transfer functions calculate a layer's output vector (or matrix) A, given its net input vector (or matrix) N. The only constraint on the relationship between the

U.S. Patent No. RE49,461	nnet

	output and net input is that the output must have the same dimensions as the input.
	
	Once defined, you can assign your transfer function to any layer of a network. For example, the following line of code assigns the transfer function `yourtf` to the second layer of a network.
	
	`net.layers{2}.transferFcn = 'yourtf';`
	
	Your transfer function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid transfer function, your function must calculate outputs `A` from net inputs `N` as follows,
	
	`A = yourtf(N)`
	
	where:
	
	• `N` is an S x Q matrix of Q net input (column) vectors.
	• `A` is an S x Q matrix of Q output (column) vectors.
	
	Your transfer function must also provide information about itself, using this calling format,
	
	`info = yourtf(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'output'` - Returns the output range.
	• `'active'` - Returns the active input range.
	
	The toolbox contains an example custom transfer function called `mytf`. Enter the following lines of code to see how it is used.
	
	`help mytf`
	`n = -5:.1:5;`
	`a = mytf(n);`
	`plot(n,a)`
	`mytf('deriv')`
	
	393.

U.S. Patent No. RE49,461	nnet

	Enter the following command to see how `mytf` is implemented.
	`type mytf`
	You can use `mytf` as a template to create your own transfer function.
	Transfer Derivative Functions. If you want to use backpropagation with your custom transfer function, you need to create a custom derivative function for it. The function needs to calculate the derivative of the layer's output with respect to its net input,
	`dA_dN = yourdtf(N,A)`
	where:
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q output (column) vectors.
	• `dA_dN` is the $S \times Q$ derivative dA/dN.
	This only works for transfer functions whose output elements are independent. In other words, where each `A(i)` is only a function of `N(i)`. Otherwise, a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom transfer derivative function `mydtf` works, type
	`help mydtf`
	`da_dn = mydtf(n,a)`
	`subplot(2,1,1), plot(n,a)`
	`subplot(2,1,2), plot(n,dn_da)`
	Use this command to see how `mydtf` was implemented.
	`type mydtf`
	You can use `mydtf` as a template to create your own transfer derivative functions.
	Net Input Functions
	Net input functions calculate a layer's net input vector (or matrix) `N`, given its weighted input vectors (or matrices) `Zi`. The only constraints on the relationship between the net input and the weighted inputs are that the net
	394.

U.S. Patent No. RE49,461	nnet

	input must have the same dimensions as the weighted inputs, and that the function cannot be sensitive to the order of the weight inputs.
	
	Once defined, you can assign your net input function to any layer of a network. For example, the following line of code assigns the transfer function `yournif` to the second layer of a network.
	
	`net.layers{2}.netInputFcn = 'yournif';`
	
	Your net input function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid net input function your function must calculate outputs `A` from net inputs `N` as follows,
	
	`N = yournif(Z1,Z2,...)`
	
	where
	
	• `Zi` is the ith $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	
	Your net input function must also provide information about itself using this calling format,
	
	`info = yournif(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	
	The toolbox contains an example custom net input function called `mynif`. Enter the following lines of code to see how it is used.
	
	`help mynif`
	`z1 = rand(4,5);`
	`z2 = rand(4,5);`
	`z3 = rand(4,5);`
	`n = mynif(z1,z2,z3)`
	`mynif('deriv')`
	
	Enter the following command to see how `mynif` is implemented.
	
	`type mynif`
	
	395.

U.S. Patent No. RE49,461	nnet

	You can use `mynif` as a template to create your own net input function.
	
	Net Input Derivative Functions. If you want to use backpropagation with your custom net input function, you need to create a custom derivative function for it. It needs to calculate the derivative of the layer's net input with respect to any of its weighted inputs,
	`dN_dZ = dtansig(Z,N)`
	where:
	• `Z` is one of the $S \times Q$ matrices of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `dN_dZ` is the $S \times Q$ derivative dN/dZ.
	To see how the example custom net input derivative function `mydtf` works, type
	`help mydnif`
	`dn_dz1 = mydnif(z1,n)`
	`dn_dz2 = mydnif(z1,n)`
	`dn_dz3 = mydnif(z1,n)`
	Use this command to see how `mydtf` was implemented.
	`type mydnif`
	You can use `mydnif` as a template to create your own net input derivative functions.
	
	Weight Functions
	Weight functions calculate a weighted input vector (or matrix) `Z`, given an input vector (or matrices) `P` and a weight matrix `W`.
	Once defined, you can assign your weight function to any input weight or layer weight of a network. For example, the following line of code assigns the weight function `yourwf` to the weight going to the second layer from the first input of a network.
	`net.inputWeights{2,1}.weightFcn = 'yourwf';`
	Your weight function is used whenever you simulate your network.
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	396.

U.S. Patent No. RE49,461	nnet

	To be a valid weight function your function must calculate weight inputs Z from inputs P and a weight matrix W as follows,
	`Z = yourwf(W,P)`
	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	Your net input function must also provide information about itself using this calling format,
	`info = yourwf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	The toolbox contains an example custom weight called mywf. Enter the following lines of code to see how it is used.
	`help mywf`
	`w = rand(1,5);`
	`p = rand(5,1);`
	`z = mywf(w,p);`
	`mywf('deriv')`
	Enter the following command to see how mywf is implemented.
	`type mywf`
	You can use mywf as a template to create your own weight functions.
	Weight Derivative Functions. If you want to use backpropagation with your custom weight function, you need to create a custom derivative function for it. It needs to calculate the derivative of the weight inputs with respect to both the input and weight,
	`dZ_dP = mydwf('p',W,P,Z)`
	`dZ_dW = mydwf('w',W,P,Z)`
	
	397.

U.S. Patent No. RE49,461	nnet

	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• dZ_dP is the $S \times R$ derivative dZ/dP.
	• dZ_dW is the $R \times Q$ derivative dZ/dW.
	This only works for weight functions whose output consists of a sum of i term, where each ith term is a function of only W(i) and P(i). Otherwise a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom net input derivative function mydwf works, type
	`help mydwf`
	`dz_dp = mydwf('p',w,p,z)`
	`dz_dw = mydwf('w',w,p,z)`
	Use this command to see how mydwf is implemented.
	`type mydwf`
	You can use mydwf as a template to create your own net input derivative function.
	Initialization Functions
	You can create three kinds of initialization functions: network, layer, and weight/bias initialization.
	Network Initialization Functions
	The most general kind of initialization function is the network initialization function which sets all the weights and biases of a network to values suitable as a starting point for training or adaption.
	Once defined, you can assign your network initialization function to a network.
	`net.initFcn = 'yournif';`
	398.

U.S. Patent No. RE49,461	nnet

	Your network initialization function is used whenever you initialize your network.
	`net = init(net)`
	To be a valid network initialization function, it must take and return a network.
	`net = yournif(net)`
	Your function can set the network's weight and bias values in any way you want. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `init` turns off the normal type checking for network properties before calling your initialization function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but could cause problems later when you try to simulate or train the network.
	You can examine the implementation of the toolbox function `initlay` if you are interested in creating your own network initialization function.
	Layer Initialization Functions
	The layer initialization function sets all the weights and biases of a layer to values suitable as a starting point for training or adaption.
	Once defined, you can assign your layer initialization function to a layer of a network. For example, the following line of code assigns the layer initialization function `yourlif` to the second layer of a network.
	`net.layers{2}.initFcn = 'yourlif';`
	Layer initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`. If this is the case, then your function is used to initialize the layer whenever you initialize your network with `init`.
	`net = init(net)`
	To be a valid layer initialization function, it must take a network and a layer index `i`, and return the network after initializing the *i*th layer.
	`net = yournif(net,i)`
	399.

U.S. Patent No. RE49,461	nnet

	Your function can then set the *i*th layer's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size.
	
	If you are interested in creating your own layer initialization function, you can examine the implementations of the toolbox functions `initwb` and `initnw`.
	
	Weight and Bias Initialization Functions
	The weight and bias initialization function sets all the weights and biases of a weight or bias to values suitable as a starting point for training or adaption.
	
	Once defined, you can assign your initialization function to any weight and bias in a network. For example, the following lines of code assign the weight and bias initialization function `yourwbif` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.initFcn = 'yourwbif';`
	`net.inputWeights{2,1}.initFcn = 'yourwbif';`
	
	Weight and bias initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`, and the layer's initialization function (`net.layers{i}.initFcn`) is set to the toolbox function `initwb`. If this is the case, then your function is used to initialize the weight and biases it is assigned to whenever you initialize your network with `init`.
	
	`net = init(net)`
	
	To be a valid weight and bias initialization function, it must take a the number of neurons in a layer `S`, and a two-column matrix `PR` of R rows defining the minimum and maximum values of R inputs and return a new weight matrix `W`,
	
	`W = rands(S,PR)`
	
	where:
	
	• `S` is the number of neurons in the layer.
	• `PR` is an $R \times 2$ matrix defining the minimum and maximum values of R inputs.
	• `W` is a new $S \times R$ weight matrix.
	
	Your function also needs to generate a new bias vector as follows,
	
	400.

U.S. Patent No. RE49,461	nnet
	b = rands(S) where: • S is the number of neurons in the layer. • b is a new $S \times 1$ bias vector. To see how an example custom weight and bias initialization function works, type help mywbif W = mywbif(4,[0 1; -2 2]) b = mywbif(4,[1 1]) Use this command to see how mywbif was implemented. type mywbif You can use mywbif as a template to create your own weight and bias initialization function. **Learning Functions** You can create four kinds of initialization functions: training, adaption, performance, and weight/bias learning. **Training Functions** One kind of general learning function is a network training function. Training functions repeatedly apply a set of input vectors to a network, updating the network each time, until some stopping criteria is met. Stopping criteria can consists of a maximum number of epochs, a minimum error gradient, an error goal, etc. Once defined, you can assign your training function to a network. net.trainFcn = 'yourtf'; Your network initialization function is used whenever you train your network. [net,tr] = train(NET,P,T,Pi,Ai) To be a valid training function your function must take and return a network. [net,tr] = yourtf(net,Pd,Tl,Ai,Q,TS,VV,TV) 401.

U.S. Patent No. RE49,461	nnet
	where: • `Pd` is an $N_l \times N_i \times$ `TS` cell array of tap delayed inputs. ▪ Each `Pd{i,j,ts}` is the $R^j \times (D_i^{ij}Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step `ts`. (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.) • `Tl` is an $N_l \times$ `TS` cell array of layer targets. ▪ Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl{i,ts}` is an empty matrix if the ith layer doesn't have a target.) • `Ai` is an $N_l \times LD$ cell array of initial layer delay states. ▪ Each `Ai{l,k}` is the $S^i \times Q$ delayed ith layer output for time step `ts =` `k`-LD, where ts goes from 0 to LD-1. • `Q` is the number of concurrent vectors. • `TS` is the number of time steps. • VV and TV are optional structures defining validation and test vectors in the same form as the training vectors defined above: `Pd`, `Tl`, `Ai`, `Q`, and `TS`. Note that the validation and testing `Q` and `TS` values can be different from each other and from those used by the training vectors. The dimensions above have the following definitions: • N_l is the number of network layers (`net.numLayers`). • N_i is the number of network inputs (`net.numInputs`). • R^j is the size of the jth input (`net.inputs{j}.size`). • S^i is the size of the ith layer (`net.layers{i}.size`) • LD is the number of layer delays (`net.numLayerDelays`). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (`length(net.inputWeights{i,j}.delays)`). Your training function must also provide information about itself using this calling format, `info = yourtf(code)` where the correct information is returned for each of the following string codes: • `'version'` - Returns the Neural Network Toolbox version (3.0). • `'pdefaults'` - Returns a structure of default training parameters. 402.

U.S. Patent No. RE49,461	nnet
	When you set the network training function (`net.trainFcn`) to be your function, the network's training parameters (`net.trainParam`) automatically is set to your default structure. Those values can be altered (or not) before training. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size. For performance reasons, `train` turns off the normal type checking for network properties before calling your training function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or adapt the network. If you are interested in creating your own training function, you can examine the implementations of toolbox functions such as `trainc` and `trainr`. The help for each of these utility functions lists the input and output arguments they take. **Utility Functions.** If you examine training functions such as `trainc`, `traingd`, and `trainlm`, note that they use a set of utility functions found in the `nnet/nnutils` directory. These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you might have to update your functions for future versions of the toolbox. Use `help` on each function to view the function's input and output arguments. These two functions are useful for creating a new training record and truncating it once the final number of epochs is known: • `newtr` - New training record with any number of optional fields. • `cliptr` - Clip training record to the final number of epochs. These three functions calculate network signals going forward, errors, and derivatives of performance coming back: • `calca` - Calculate network outputs and other signals. • `calcerr` - Calculate matrix or cell array errors. • `calcgrad` - Calculate bias and weight performance gradients. 403.

U.S. Patent No. RE49,461	nnet

	These two functions get and set a network's weight and bias values with single vectors. Being able to treat all these adjustable parameters as a single vector is often useful for implementing optimization algorithms:
	• `getx` - Get all network weight and bias values as a single vector.
	• `setx` - Set all network weight and bias values with a single vector.
	These next three functions are also useful for implementing optimization functions. One calculates all network signals going forward, including errors and performance. One backpropagates to find the derivatives of performance as a single vector. The third function backpropagates to find the Jacobian of performance. This latter function is used by advanced optimization techniques like Levenberg-Marquardt:
	• `calcperf` - Calculate network outputs, signals, and performance.
	• `calcgx` - Calculate weight and bias performance gradient as a single vector.
	• `calcjx` - Calculate weight and bias performance Jacobian as a single matrix.
	Adapt Functions
	The other kind of the general learning function is a network adapt function. Adapt functions simulate a network, while updating the network for each time step of the input before continuing the simulation to the next input.
	Once defined, you can assign your adapt function to a network.
	`net.adaptFcn = 'youraf';`
	Your network initialization function is used whenever you adapt your network.
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid adapt function, it must take and return a network,
	`[net,Ac,El] = youraf(net,Pd,Tl,Ai,Q,TS)`
	where:
	• `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
	- Each `Pd{i,j,ts}` is the $R^j \times (D_i{}^{ij}Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step `ts`. Note that (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.)
	404.

U.S. Patent No. RE49,461	nnet
	• `Tl` is an $N_l \times TS$ cell array of layer targets. - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. Note that (`Tl{i,ts}` is an empty matrix if the ith layer doesn't have a target.) • `Ai` is an $N_l \times LD$ cell array of initial layer delay states. - Each `Ai{l,k}` is the $S^i \times Q$ delayed ith layer output for time step `ts = k-`LD, where ts goes from 0 to LD-1. • `Q` is the number of concurrent vectors. • `TS` is the number of time steps. The dimensions above have the following definitions: • N_l is the number of network layers (`net.numLayers`). • N_i is the number of network inputs (`net.numInputs`). • R^j is the size of the jth input (`net.inputs{j}.size`). • S^i is the size of the ith layer (`net.layers{i}.size`) • LD is the number of layer delays (`net.numLayerDelays`). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (`length(net.inputWeights{i,j}.delays)`). Your adapt function must also provide information about itself using this calling format, `info = youraf(code)` where the correct information is returned for each of the following string codes: • `'version'` - Returns the Neural Network Toolbox version (3.0). • `'pdefaults'` - Returns a structure of default adapt parameters. When you set the network adapt function (`net.adaptFcn`) to be your function, the network's adapt parameters (`net.adaptParam`) automatically is set to your default structure. Those values can then be altered (or not) before adapting. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `adapt` turns off the normal type checking for network properties before calling your adapt function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or train the network. 405.

U.S. Patent No. RE49,461	nnet

	If you are interested in creating your own training function, you can examine the implementation of a toolbox function such as `trains`.
	
	Utility Functions. If you examine the toolbox's only adapt function `trains`, note that it uses a set of utility functions found in the `nnet/nnutils` directory. The help for each of these utility functions lists the input and output arguments they take.
	
	These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you will have to update your functions for future versions of the toolbox.
	
	These two functions are useful for simulating a network, and calculating its derivatives of performance:
	
	• `calca1` - New training record with any number of optional fields.
	• `calce1` - Clip training record to the final number of epochs.
	• `calcgrad` - Calculate bias and weight performance gradients.
	
	Performance Functions
	Performance functions allow a network's behavior to be graded. This is useful for many algorithms, such as backpropagation, which operate by adjusting network weights and biases to improve performance.
	
	Once defined you can assign your training function to a network.
	
	`net.performFcn = 'yourpf';`
	
	Your network initialization function will then be used whenever you train your adapt your network.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid performance function your function must be called as follows,
	
	`perf = yourpf(E,X,PP)`
	
	where:
	
	• `E` is either an S x Q matrix or an $N_l \times TS$ cell array of layer errors.
	
	406.

U.S. Patent No. RE49,461	nnet

	- Each `E{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl(i,ts)` is an empty matrix if the ith layer doesn't have a target.)
	• `X` is an M x 1 vector of all the network's weights and biases.
	• `PP` is a structure of network performance parameters.
	If E is a cell array you can convert it to a matrix as follows.
	`E = cell2mat(E);`
	Alternatively, your function must also be able to be called as follows,
	`perf = yourpf(E,net)`
	where you can get `X` and `PP` (if needed) as follows.
	`X = getx(net);`
	`PP = net.performParam;`
	Your performance function must also provide information about itself using this calling format,
	`info = yourpf(code)`
	where the correct information is returned for each of the following string codes:
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'pdefaults'` - Returns a structure of default performance parameters.
	When you set the network performance function (`net.performFcn`) to be your function, the network's adapt parameters (`net.performParam`) will automatically get set to your default structure. Those values can then be altered or not before training or adaption.
	To see how an example custom performance function works type in these lines of code.
	`help mypf`
	`e = rand(4,5);`
	`x = rand(12,1);`
	`pp = mypf('pdefaults')`
	`perf = mypf(e,x,pp)`
	
	407.

U.S. Patent No. RE49,461	nnet

	Use this command to see how mypf was implemented.
	`type mypf`
	You can use mypf as a template to create your own weight and bias initialization function.
	Performance Derivative Functions. If you want to use backpropagation with your performance function, you need to create a custom derivative function for it. It needs to calculate the derivative of the network's errors and combined weight and bias vector, with respect to performance,
	`dPerf_dE = dmsereg('e',E,X,perf,PP)`
	`dPerf_dX = dmsereg('x',E,X,perf,PP)`
	where:
	• E is an $N_l \times TS$ cell array of layer errors.
	- Each E{i,ts} is the $S^i \times Q$ target matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• X is an $M \times 1$ vector of all the network's weights and biases.
	• PP is a structure of network performance parameters.
	• dPerf_dE is the $N_l \times TS$ cell array of derivatives dPerf/dE.
	- Each E{i,ts} is the $S^i \times Q$ derivative matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• dPerf_dX is the $M \times 1$ derivative dPerf/dX.
	To see how the example custom performance derivative function mydpf works, type
	`help mydpf`
	`e = {e};`
	`dperf_de = mydpf('e',e,x,perf,pp)`
	`dperf_dx = mydpf('x',e,x,perf,pp)`
	Use this command to see how mydpf was implemented.
	`type mydpf`
	You can use mydpf as a template to create your own performance derivative functions.
	408.

U.S. Patent No. RE49,461	nnet

	Weight and Bias Learning Functions
	The most specific kind of learning function is a weight and bias learning function. These functions are used to update individual weights and biases during learning. with some training and adapt functions.
	
	Once defined. you can assign your learning function to any weight and bias in a network. For example, the following lines of code assign the weight and bias learning function `yourwblf` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.learnFcn = 'yourwblf';`
	`net.inputWeights{2,1}.learnFcn = 'yourwblf';`
	
	Weight and bias learning functions are only called to update weights and biases if the network training function (`net.trainFcn`) is set to `trainb`, `trainc`, or `trainr`, or if the network adapt function (`net.adaptFcn`) is set to `trains`. If this is the case, then your function is used to update the weight and biases it is assigned to whenever you train or adapt your network with `train` or `adapt`.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid weight and bias learning function, it must be callable as follows,
	
	`[dW,LS] = yourwblf(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)`
	
	where:
	
	• `W` is an $S \times R$ weight matrix.
	• `P` is an $R \times Q$ matrix of Q input (column) vectors.
	• `Z` is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q layer output (column) vectors.
	• `T` is an $S \times Q$ matrix of Q target (column) vectors.
	• `E` is an $S \times Q$ matrix of Q error (column) vectors.
	• `gW` is an $S \times R$ gradient of `W` with respect to performance.
	• `gA` is an $S \times Q$ gradient of `A` with respect to performance.
	• `D` is an $S \times S$ matrix of neuron distances.
	• `LP` is a a structure of learning parameters.
	
	409.

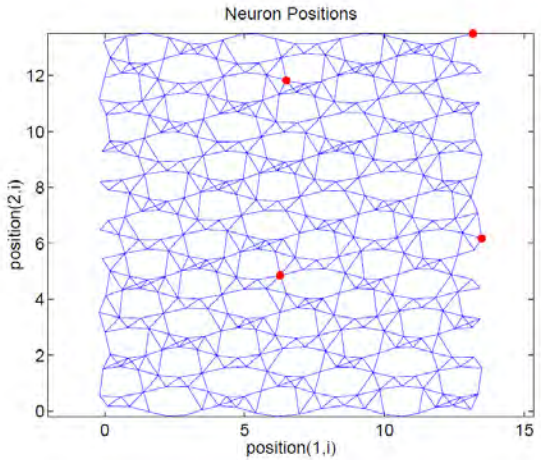
U.S. Patent No. RE49,461	nnet

	• LS is a structure of the learning state that is updated for each call. (Use a null matrix [ ] the first time.)
	• dW is the resulting $S \times R$ weight change matrix.
	Your function is called as follows to update bias vector
	`[db,LS] = yourwblf(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)`
	where:
	• S is the number of neurons in the layer.
	• b is a new $S \times 1$ bias vector.
	Your learning function must also provide information about itself using this calling format,
	`info = yourwblf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	• 'pdefaults' - Returns a structure of default performance parameters.
	To see how an example custom weight and bias initialization function works, type
	`help mywblf`
	Use this command to see how mywbif was implemented.
	`type mywblf`
	You can use mywblf as a template to create your own weight and bias learning function.
	Self-Organizing Map Functions
	There are two kinds of functions that control how neurons in self-organizing maps respond. They are topology and distance functions.
	Topology Functions
	Topology functions calculate the positions of a layer's neurons given its dimensions.
	
	410.

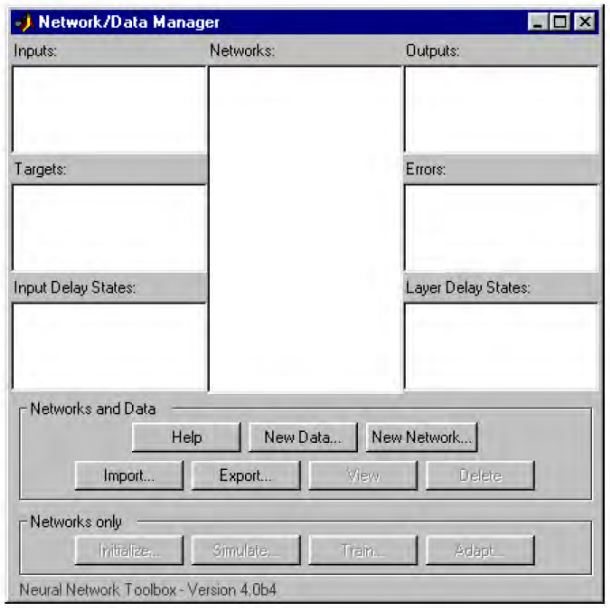
U.S. Patent No. RE49,461	nnet

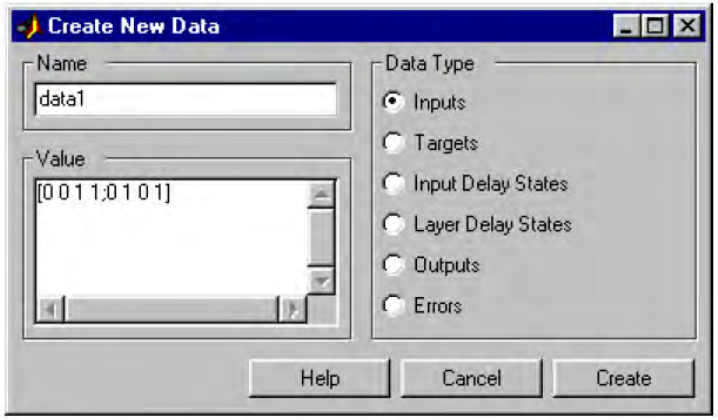
	Once defined, you can assign your topology function to any layer of a network. For example, the following line of code assigns the topology function `yourtopf` to the second layer of a network.
	`net.layers{2}.topologyFcn = 'yourtopf';`
	Your topology function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid topology function your function must calculate positions `pos` from dimensions `dim` as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• `dimi` is the number of neurons along the ith dimension of the layer.
	• `pos` is an $N \times S$ matrix of S position vectors, where S is the total number of neurons that is defined by the product `dim1*dim1*...*dimN`.
	The toolbox contains an example custom topology function called `mytopf`. Enter the following lines of code to see how it is used.
	`help mytopf`
	`pos = mytopf(20,20);`
	`plotsom(pos)`
	If you type that code, you get the following plot.
	
	411.

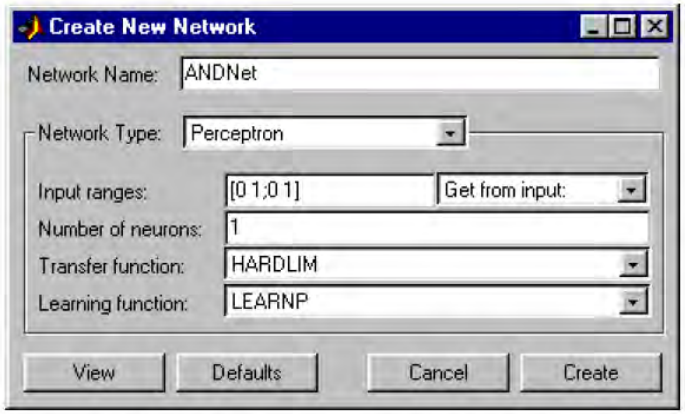
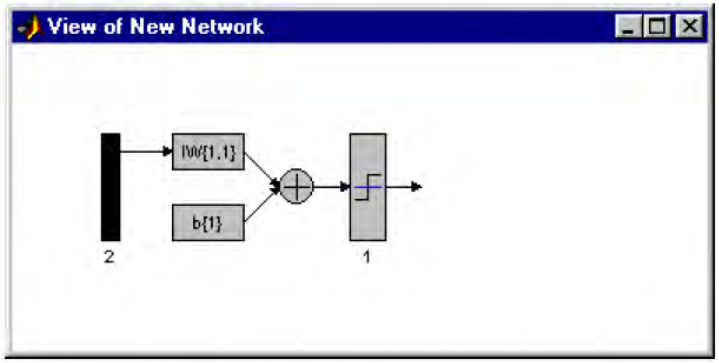
U.S. Patent No. RE49,461	nnet

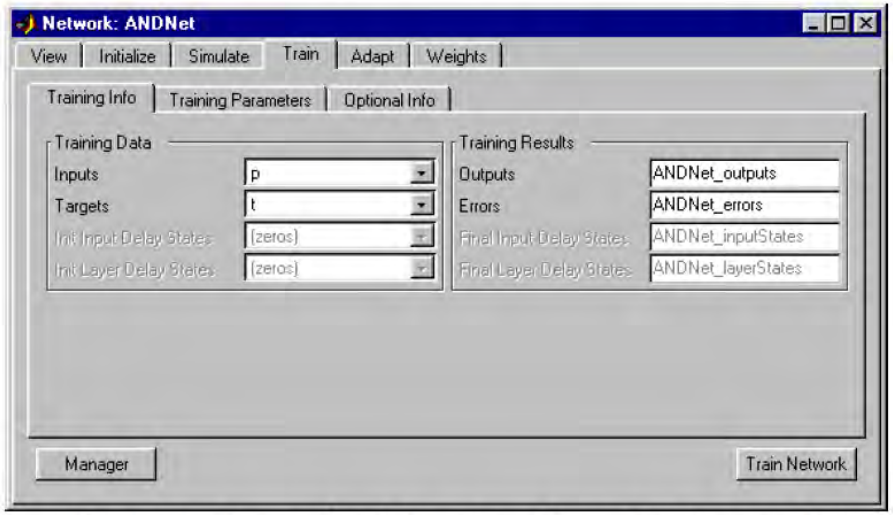
	
	Enter the following command to see how `mytf` is implemented.
	`type mytopf`
	You can use `mytopf` as a template to create your own topology function.
	Distance Functions
	Distance functions calculate the distances of a layer's neurons given their position.
	Once defined, you can assign your distance function to any layer of a network. For example, the following line of code assigns the topology function `yourdistf` to the second layer of a network.
	`net.layers{2}.distanceFcn = 'yourdistf';`
	Your distance function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	412.

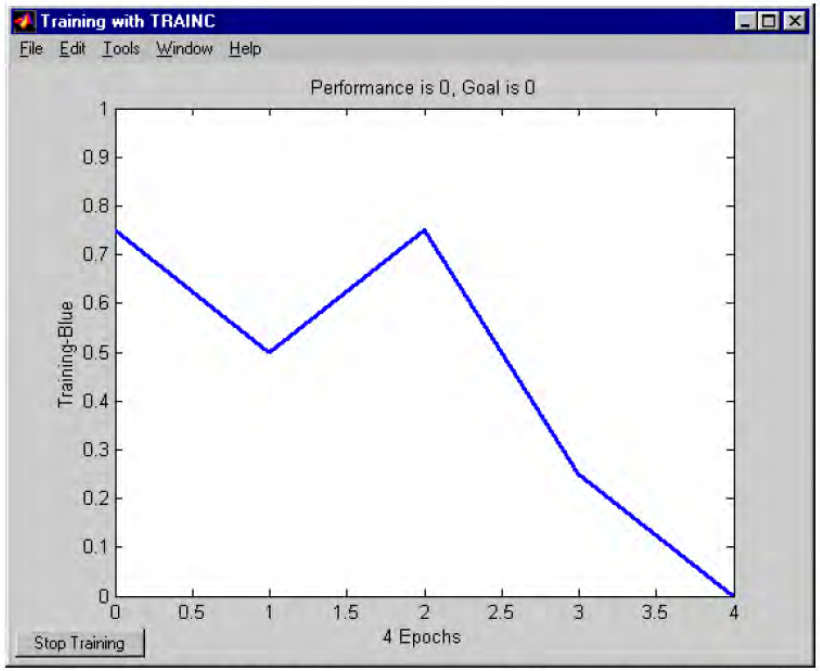
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
23. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of computational elements of the computations representing the artificial neural network.	nnet discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of computational elements of the computations representing the artificial neural network. *See* claim 21; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses controlling setting and editing of computational elements of the computations representing the artificial neural network (e.g., nntool, custom functions).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
   2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
   ans =
   -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

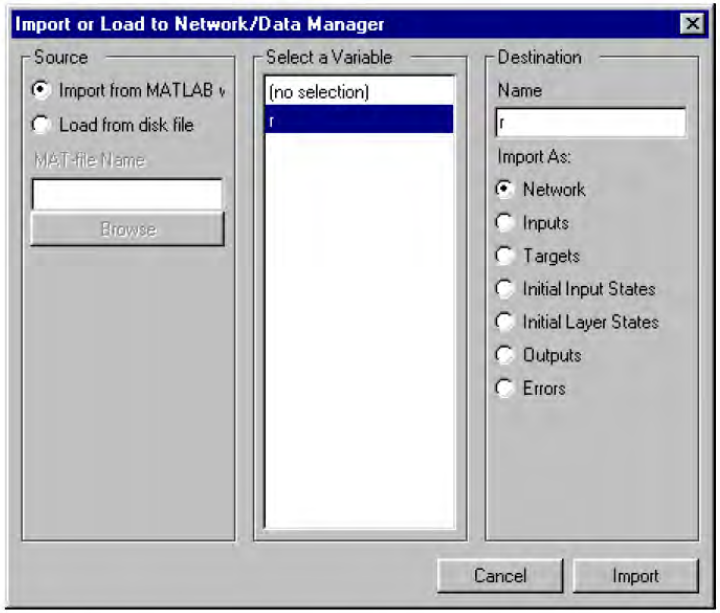
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

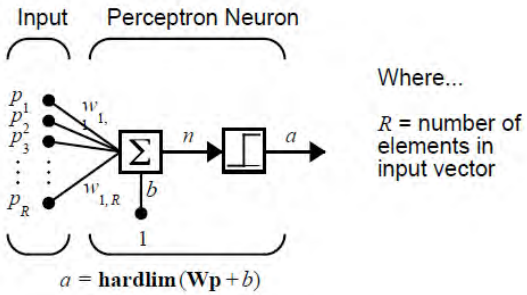
Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

U.S. Patent No. RE49,461	nnet
	392.

12 Advanced Topics

Custom Functions

The toolbox allows you to create and use many kinds of functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for your networks.

The following sections describe how to create your own versions of these kinds of functions:

- Simulation functions
  - transfer functions
  - net input functions
  - weight functions
- Initialization functions
  - network initialization functions
  - layer initialization functions
  - weight and bias initialization functions
- Learning functions
  - network training functions
  - network adapt functions
  - network performance functions
  - weight and bias learning functions
- Self-organizing map functions
  - topology functions
  - distance functions

Simulation Functions

You can create three kinds of simulation functions: transfer, net input, and weight functions. You can also provide associated derivative functions to enable backpropagation learning with your functions.

Transfer Functions

Transfer functions calculate a layer's output vector (or matrix) A, given its net input vector (or matrix) N. The only constraint on the relationship between the

U.S. Patent No. RE49,461	nnet

	output and net input is that the output must have the same dimensions as the input.
	
	Once defined, you can assign your transfer function to any layer of a network. For example, the following line of code assigns the transfer function `yourtf` to the second layer of a network.
	
	`net.layers{2}.transferFcn = 'yourtf';`
	
	Your transfer function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid transfer function, your function must calculate outputs `A` from net inputs `N` as follows,
	
	`A = yourtf(N)`
	
	where:
	
	• `N` is an S x Q matrix of Q net input (column) vectors.
	• `A` is an S x Q matrix of Q output (column) vectors.
	
	Your transfer function must also provide information about itself, using this calling format,
	
	`info = yourtf(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'output'` - Returns the output range.
	• `'active'` - Returns the active input range.
	
	The toolbox contains an example custom transfer function called `mytf`. Enter the following lines of code to see how it is used.
	
	`help mytf`
	`n = -5:.1:5;`
	`a = mytf(n);`
	`plot(n,a)`
	`mytf('deriv')`
	
	393.

U.S. Patent No. RE49,461	nnet

	Enter the following command to see how `mytf` is implemented.
	`type mytf`
	You can use `mytf` as a template to create your own transfer function.
	Transfer Derivative Functions. If you want to use backpropagation with your custom transfer function, you need to create a custom derivative function for it. The function needs to calculate the derivative of the layer's output with respect to its net input,
	`dA_dN = yourdtf(N,A)`
	where:
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q output (column) vectors.
	• `dA_dN` is the $S \times Q$ derivative dA/dN.
	This only works for transfer functions whose output elements are independent. In other words, where each `A(i)` is only a function of `N(i)`. Otherwise, a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom transfer derivative function `mydtf` works, type
	`help mydtf`
	`da_dn = mydtf(n,a)`
	`subplot(2,1,1), plot(n,a)`
	`subplot(2,1,2), plot(n,dn_da)`
	Use this command to see how `mydtf` was implemented.
	`type mydtf`
	You can use `mydtf` as a template to create your own transfer derivative functions.
	Net Input Functions
	Net input functions calculate a layer's net input vector (or matrix) `N`, given its weighted input vectors (or matrices) `Zi`. The only constraints on the relationship between the net input and the weighted inputs are that the net
	394.

U.S. Patent No. RE49,461	nnet

	input must have the same dimensions as the weighted inputs, and that the function cannot be sensitive to the order of the weight inputs.
	
	Once defined, you can assign your net input function to any layer of a network. For example, the following line of code assigns the transfer function `yournif` to the second layer of a network.
	
	`net.layers{2}.netInputFcn = 'yournif';`
	
	Your net input function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid net input function your function must calculate outputs `A` from net inputs `N` as follows,
	
	`N = yournif(Z1,Z2,...)`
	
	where
	
	• `Zi` is the ith $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	
	Your net input function must also provide information about itself using this calling format,
	
	`info = yournif(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	The toolbox contains an example custom net input function called `mynif`. Enter the following lines of code to see how it is used.
	
	`help mynif`
	`z1 = rand(4,5);`
	`z2 = rand(4,5);`
	`z3 = rand(4,5);`
	`n = mynif(z1,z2,z3)`
	`mynif('deriv')`
	
	Enter the following command to see how `mynif` is implemented.
	
	`type mynif`
	
	395.

U.S. Patent No. RE49,461	nnet

	You can use `mynif` as a template to create your own net input function.
	
	Net Input Derivative Functions. If you want to use backpropagation with your custom net input function, you need to create a custom derivative function for it. It needs to calculate the derivative of the layer's net input with respect to any of its weighted inputs,
	`dN_dZ = dtansig(Z,N)`
	where:
	• `Z` is one of the $S \times Q$ matrices of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `dN_dZ` is the $S \times Q$ derivative dN/dZ.
	To see how the example custom net input derivative function `mydtf` works, type
	`help mydnif`
	`dn_dz1 = mydnif(z1,n)`
	`dn_dz2 = mydnif(z1,n)`
	`dn_dz3 = mydnif(z1,n)`
	Use this command to see how `mydtf` was implemented.
	`type mydnif`
	You can use `mydnif` as a template to create your own net input derivative functions.
	
	Weight Functions
	Weight functions calculate a weighted input vector (or matrix) `Z`, given an input vector (or matrices) `P` and a weight matrix `W`.
	Once defined, you can assign your weight function to any input weight or layer weight of a network. For example, the following line of code assigns the weight function `yourwf` to the weight going to the second layer from the first input of a network.
	`net.inputWeights{2,1}.weightFcn = 'yourwf';`
	Your weight function is used whenever you simulate your network.
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	396.

U.S. Patent No. RE49,461	nnet

	To be a valid weight function your function must calculate weight inputs Z from inputs P and a weight matrix W as follows,
	`Z = yourwf(W,P)`
	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	Your net input function must also provide information about itself using this calling format,
	`info = yourwf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	The toolbox contains an example custom weight called mywf. Enter the following lines of code to see how it is used.
	`help mywf`
	`w = rand(1,5);`
	`p = rand(5,1);`
	`z = mywf(w,p);`
	`mywf('deriv')`
	Enter the following command to see how mywf is implemented.
	`type mywf`
	You can use mywf as a template to create your own weight functions.
	Weight Derivative Functions. If you want to use backpropagation with your custom weight function, you need to create a custom derivative function for it. It needs to calculate the derivative of the weight inputs with respect to both the input and weight,
	`dZ_dP = mydwf('p',W,P,Z)`
	`dZ_dW = mydwf('w',W,P,Z)`
	
	397.

U.S. Patent No. RE49,461	nnet

	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• dZ_dP is the $S \times R$ derivative dZ/dP.
	• dZ_dW is the $R \times Q$ derivative dZ/dW.
	This only works for weight functions whose output consists of a sum of i term, where each ith term is a function of only W(i) and P(i). Otherwise a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom net input derivative function mydwf works, type
	`help mydwf`
	`dz_dp = mydwf('p',w,p,z)`
	`dz_dw = mydwf('w',w,p,z)`
	Use this command to see how mydwf is implemented.
	`type mydwf`
	You can use mydwf as a template to create your own net input derivative function.
	Initialization Functions
	You can create three kinds of initialization functions: network, layer, and weight/bias initialization.
	Network Initialization Functions
	The most general kind of initialization function is the network initialization function which sets all the weights and biases of a network to values suitable as a starting point for training or adaption.
	Once defined, you can assign your network initialization function to a network.
	`net.initFcn = 'yournif';`
	398.

U.S. Patent No. RE49,461	nnet

	Your network initialization function is used whenever you initialize your network.
	`net = init(net)`
	To be a valid network initialization function, it must take and return a network.
	`net = yournif(net)`
	Your function can set the network's weight and bias values in any way you want. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `init` turns off the normal type checking for network properties before calling your initialization function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but could cause problems later when you try to simulate or train the network.
	You can examine the implementation of the toolbox function `initlay` if you are interested in creating your own network initialization function.
	Layer Initialization Functions
	The layer initialization function sets all the weights and biases of a layer to values suitable as a starting point for training or adaption.
	Once defined, you can assign your layer initialization function to a layer of a network. For example, the following line of code assigns the layer initialization function `yourlif` to the second layer of a network.
	`net.layers{2}.initFcn = 'yourlif';`
	Layer initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`. If this is the case, then your function is used to initialize the layer whenever you initialize your network with `init`.
	`net = init(net)`
	To be a valid layer initialization function, it must take a network and a layer index `i`, and return the network after initializing the *i*th layer.
	`net = yournif(net,i)`
	399.

U.S. Patent No. RE49,461	nnet

	Your function can then set the *i*th layer's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size.
	
	If you are interested in creating your own layer initialization function, you can examine the implementations of the toolbox functions `initwb` and `initnw`.
	
	Weight and Bias Initialization Functions
	The weight and bias initialization function sets all the weights and biases of a weight or bias to values suitable as a starting point for training or adaption.
	
	Once defined, you can assign your initialization function to any weight and bias in a network. For example, the following lines of code assign the weight and bias initialization function `yourwbif` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.initFcn = 'yourwbif';`
	`net.inputWeights{2,1}.initFcn = 'yourwbif';`
	
	Weight and bias initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`, and the layer's initialization function (`net.layers{i}.initFcn`) is set to the toolbox function `initwb`. If this is the case, then your function is used to initialize the weight and biases it is assigned to whenever you initialize your network with `init`.
	
	`net = init(net)`
	
	To be a valid weight and bias initialization function, it must take a the number of neurons in a layer `S`, and a two-column matrix `PR` of R rows defining the minimum and maximum values of R inputs and return a new weight matrix `W`,
	
	`W = rands(S,PR)`
	
	where:
	
	• `S` is the number of neurons in the layer.
	• `PR` is an $R \times 2$ matrix defining the minimum and maximum values of R inputs.
	• `W` is a new $S \times R$ weight matrix.
	
	Your function also needs to generate a new bias vector as follows,
	
	400.

U.S. Patent No. RE49,461	nnet
	b = rands(S) where: • S is the number of neurons in the layer. • b is a new $S \times 1$ bias vector. To see how an example custom weight and bias initialization function works, type help mywbif W = mywbif(4,[0 1; -2 2]) b = mywbif(4,[1 1]) Use this command to see how mywbif was implemented. type mywbif You can use mywbif as a template to create your own weight and bias initialization function. **Learning Functions** You can create four kinds of initialization functions: training, adaption, performance, and weight/bias learning. **Training Functions** One kind of general learning function is a network training function. Training functions repeatedly apply a set of input vectors to a network, updating the network each time, until some stopping criteria is met. Stopping criteria can consists of a maximum number of epochs, a minimum error gradient, an error goal, etc. Once defined, you can assign your training function to a network. net.trainFcn = 'yourtf'; Your network initialization function is used whenever you train your network. [net,tr] = train(NET,P,T,Pi,Ai) To be a valid training function your function must take and return a network. [net,tr] = yourtf(net,Pd,Tl,Ai,Q,TS,VV,TV) 401.

U.S. Patent No. RE49,461	nnet
	where: • `Pd` is an $N_l \times N_i \times TS$ cell array of tap delayed inputs. ▪ Each `Pd{i,j,ts}` is the $R^j \times (D_i^{ij} Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step `ts`. (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.) • `Tl` is an $N_l \times TS$ cell array of layer targets. ▪ Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl{i,ts}` is an empty matrix if the ith layer doesn't have a target.) • `Ai` is an $N_l \times LD$ cell array of initial layer delay states. ▪ Each `Ai{l,k}` is the $S^i \times Q$ delayed ith layer output for time step `ts` = `k`-LD, where ts goes from 0 to LD-1. • `Q` is the number of concurrent vectors. • `TS` is the number of time steps. • VV and TV are optional structures defining validation and test vectors in the same form as the training vectors defined above: `Pd`, `Tl`, `Ai`, `Q`, and `TS`. Note that the validation and testing `Q` and `TS` values can be different from each other and from those used by the training vectors. The dimensions above have the following definitions: • N_l is the number of network layers (`net.numLayers`). • N_i is the number of network inputs (`net.numInputs`). • R^j is the size of the jth input (`net.inputs{j}.size`). • S^i is the size of the ith layer (`net.layers{i}.size`) • LD is the number of layer delays (`net.numLayerDelays`). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (`length(net.inputWeights{i,j}.delays)`). Your training function must also provide information about itself using this calling format, `info = yourtf(code)` where the correct information is returned for each of the following string codes: • `'version'` - Returns the Neural Network Toolbox version (3.0). • `'pdefaults'` - Returns a structure of default training parameters. 402.

U.S. Patent No. RE49,461	nnet
	When you set the network training function (`net.trainFcn`) to be your function, the network's training parameters (`net.trainParam`) automatically is set to your default structure. Those values can be altered (or not) before training. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size. For performance reasons, `train` turns off the normal type checking for network properties before calling your training function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or adapt the network. If you are interested in creating your own training function, you can examine the implementations of toolbox functions such as `trainc` and `trainr`. The help for each of these utility functions lists the input and output arguments they take. **Utility Functions.** If you examine training functions such as `trainc`, `traingd`, and `trainlm`, note that they use a set of utility functions found in the `nnet/nnutils` directory. These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you might have to update your functions for future versions of the toolbox. Use `help` on each function to view the function's input and output arguments. These two functions are useful for creating a new training record and truncating it once the final number of epochs is known: • `newtr` - New training record with any number of optional fields. • `cliptr` - Clip training record to the final number of epochs. These three functions calculate network signals going forward, errors, and derivatives of performance coming back: • `calca` - Calculate network outputs and other signals. • `calcerr` - Calculate matrix or cell array errors. • `calcgrad` - Calculate bias and weight performance gradients. 403.

U.S. Patent No. RE49,461	nnet

	These two functions get and set a network's weight and bias values with single vectors. Being able to treat all these adjustable parameters as a single vector is often useful for implementing optimization algorithms:
	• `getx` - Get all network weight and bias values as a single vector.
	• `setx` - Set all network weight and bias values with a single vector.
	These next three functions are also useful for implementing optimization functions. One calculates all network signals going forward, including errors and performance. One backpropagates to find the derivatives of performance as a single vector. The third function backpropagates to find the Jacobian of performance. This latter function is used by advanced optimization techniques like Levenberg-Marquardt:
	• `calcperf` - Calculate network outputs, signals, and performance.
	• `calcgx` - Calculate weight and bias performance gradient as a single vector.
	• `calcjx` - Calculate weight and bias performance Jacobian as a single matrix.
	Adapt Functions
	The other kind of the general learning function is a network adapt function. Adapt functions simulate a network, while updating the network for each time step of the input before continuing the simulation to the next input.
	Once defined, you can assign your adapt function to a network.
	`net.adaptFcn = 'youraf';`
	Your network initialization function is used whenever you adapt your network.
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid adapt function, it must take and return a network,
	`[net,Ac,El] = youraf(net,Pd,Tl,Ai,Q,TS)`
	where:
	• `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
	- Each `Pd{i,j,ts}` is the $R^j \times (D_i{}^{ij}Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step `ts`. Note that (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.)
	404.

U.S. Patent No. RE49,461	nnet
	• `Tl` is an $N_l \times TS$ cell array of layer targets. - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. Note that (`Tl{i,ts}` is an empty matrix if the ith layer doesn't have a target.) • `Ai` is an $N_l \times LD$ cell array of initial layer delay states. - Each `Ai{l,k}` is the $S^i \times Q$ delayed ith layer output for time step `ts = k`-LD, where ts goes from 0 to LD-1. • `Q` is the number of concurrent vectors. • `TS` is the number of time steps. The dimensions above have the following definitions: • N_l is the number of network layers (`net.numLayers`). • N_i is the number of network inputs (`net.numInputs`). • R^j is the size of the jth input (`net.inputs{j}.size`). • S^i is the size of the ith layer (`net.layers{i}.size`) • LD is the number of layer delays (`net.numLayerDelays`). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (`length(net.inputWeights{i,j}.delays)`). Your adapt function must also provide information about itself using this calling format, `info = youraf(code)` where the correct information is returned for each of the following string codes: • `'version'` - Returns the Neural Network Toolbox version (3.0). • `'pdefaults'` - Returns a structure of default adapt parameters. When you set the network adapt function (`net.adaptFcn`) to be your function, the network's adapt parameters (`net.adaptParam`) automatically is set to your default structure. Those values can then be altered (or not) before adapting. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `adapt` turns off the normal type checking for network properties before calling your adapt function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or train the network. 405.

U.S. Patent No. RE49,461	nnet

	If you are interested in creating your own training function, you can examine the implementation of a toolbox function such as `trains`.
	
	Utility Functions. If you examine the toolbox's only adapt function `trains`, note that it uses a set of utility functions found in the `nnet/nnutils` directory. The help for each of these utility functions lists the input and output arguments they take.
	
	These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you will have to update your functions for future versions of the toolbox.
	
	These two functions are useful for simulating a network, and calculating its derivatives of performance:
	
	• `calca1` - New training record with any number of optional fields.
	• `calce1` - Clip training record to the final number of epochs.
	• `calcgrad` - Calculate bias and weight performance gradients.
	
	Performance Functions
	Performance functions allow a network's behavior to be graded. This is useful for many algorithms, such as backpropagation, which operate by adjusting network weights and biases to improve performance.
	
	Once defined you can assign your training function to a network.
	
	`net.performFcn = 'yourpf';`
	
	Your network initialization function will then be used whenever you train your adapt your network.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid performance function your function must be called as follows,
	
	`perf = yourpf(E,X,PP)`
	
	where:
	
	• `E` is either an S x Q matrix or an $N_l \times TS$ cell array of layer errors.
	
	406.

U.S. Patent No. RE49,461	nnet

	- Each `E{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl(i,ts)` is an empty matrix if the ith layer doesn't have a target.)
	• `X` is an M x 1 vector of all the network's weights and biases.
	• `PP` is a structure of network performance parameters.
	If E is a cell array you can convert it to a matrix as follows.
	`E = cell2mat(E);`
	Alternatively, your function must also be able to be called as follows,
	`perf = yourpf(E,net)`
	where you can get `X` and `PP` (if needed) as follows.
	`X = getx(net);`
	`PP = net.performParam;`
	Your performance function must also provide information about itself using this calling format,
	`info = yourpf(code)`
	where the correct information is returned for each of the following string codes:
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'pdefaults'` - Returns a structure of default performance parameters.
	When you set the network performance function (`net.performFcn`) to be your function, the network's adapt parameters (`net.performParam`) will automatically get set to your default structure. Those values can then be altered or not before training or adaption.
	To see how an example custom performance function works type in these lines of code.
	`help mypf`
	`e = rand(4,5);`
	`x = rand(12,1);`
	`pp = mypf('pdefaults')`
	`perf = mypf(e,x,pp)`
	
	407.

U.S. Patent No. RE49,461	nnet

	Use this command to see how mypf was implemented.
	`type mypf`
	You can use mypf as a template to create your own weight and bias initialization function.
	Performance Derivative Functions. If you want to use backpropagation with your performance function, you need to create a custom derivative function for it. It needs to calculate the derivative of the network's errors and combined weight and bias vector, with respect to performance,
	`dPerf_dE = dmsereg('e',E,X,perf,PP)`
	`dPerf_dX = dmsereg('x',E,X,perf,PP)`
	where:
	• E is an $N_l \times TS$ cell array of layer errors.
	- Each E{i,ts} is the $S^i \times Q$ target matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• X is an $M \times 1$ vector of all the network's weights and biases.
	• PP is a structure of network performance parameters.
	• dPerf_dE is the $N_l \times TS$ cell array of derivatives dPerf/dE.
	- Each E{i,ts} is the $S^i \times Q$ derivative matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• dPerf_dX is the $M \times 1$ derivative dPerf/dX.
	To see how the example custom performance derivative function mydpf works, type
	`help mydpf`
	`e = {e};`
	`dperf_de = mydpf('e',e,x,perf,pp)`
	`dperf_dx = mydpf('x',e,x,perf,pp)`
	Use this command to see how mydpf was implemented.
	`type mydpf`
	You can use mydpf as a template to create your own performance derivative functions.
	408.

U.S. Patent No. RE49,461	nnet

	Weight and Bias Learning Functions
	The most specific kind of learning function is a weight and bias learning function. These functions are used to update individual weights and biases during learning. with some training and adapt functions.
	
	Once defined. you can assign your learning function to any weight and bias in a network. For example, the following lines of code assign the weight and bias learning function `yourwblf` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.learnFcn = 'yourwblf';`
	`net.inputWeights{2,1}.learnFcn = 'yourwblf';`
	
	Weight and bias learning functions are only called to update weights and biases if the network training function (`net.trainFcn`) is set to `trainb`, `trainc`, or `trainr`, or if the network adapt function (`net.adaptFcn`) is set to `trains`. If this is the case, then your function is used to update the weight and biases it is assigned to whenever you train or adapt your network with `train` or `adapt`.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid weight and bias learning function, it must be callable as follows,
	
	`[dW,LS] = yourwblf(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)`
	
	where:
	
	• `W` is an $S \times R$ weight matrix.
	• `P` is an $R \times Q$ matrix of Q input (column) vectors.
	• `Z` is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q layer output (column) vectors.
	• `T` is an $S \times Q$ matrix of Q target (column) vectors.
	• `E` is an $S \times Q$ matrix of Q error (column) vectors.
	• `gW` is an $S \times R$ gradient of `W` with respect to performance.
	• `gA` is an $S \times Q$ gradient of `A` with respect to performance.
	• `D` is an $S \times S$ matrix of neuron distances.
	• `LP` is a a structure of learning parameters.
	
	409.

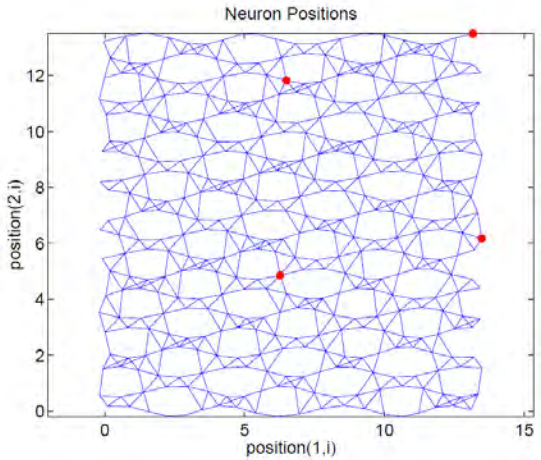
U.S. Patent No. RE49,461	nnet

	• LS is a structure of the learning state that is updated for each call. (Use a null matrix [] the first time.)
	• dW is the resulting $S \times R$ weight change matrix.
	
	Your function is called as follows to update bias vector
	
	`[db,LS] = yourwblf(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)`
	
	where:
	
	• S is the number of neurons in the layer.
	• b is a new $S \times 1$ bias vector.
	
	Your learning function must also provide information about itself using this calling format,
	
	`info = yourwblf(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	• 'pdefaults' - Returns a structure of default performance parameters.
	To see how an example custom weight and bias initialization function works, type
	
	`help mywblf`
	
	Use this command to see how mywbif was implemented.
	
	`type mywblf`
	
	You can use mywblf as a template to create your own weight and bias learning function.
	
	Self-Organizing Map Functions
	There are two kinds of functions that control how neurons in self-organizing maps respond. They are topology and distance functions.
	
	Topology Functions
	Topology functions calculate the positions of a layer's neurons given its dimensions.
	
	410.

U.S. Patent No. RE49,461	nnet

	Once defined, you can assign your topology function to any layer of a network. For example, the following line of code assigns the topology function `yourtopf` to the second layer of a network.
	`net.layers{2}.topologyFcn = 'yourtopf';`
	Your topology function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid topology function your function must calculate positions `pos` from dimensions `dim` as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• `dimi` is the number of neurons along the ith dimension of the layer.
	• `pos` is an $N \times S$ matrix of S position vectors, where S is the total number of neurons that is defined by the product `dim1*dim1*...*dimN`.
	The toolbox contains an example custom topology function called `mytopf`. Enter the following lines of code to see how it is used.
	`help mytopf`
	`pos = mytopf(20,20);`
	`plotsom(pos)`
	If you type that code, you get the following plot.
	
	411.

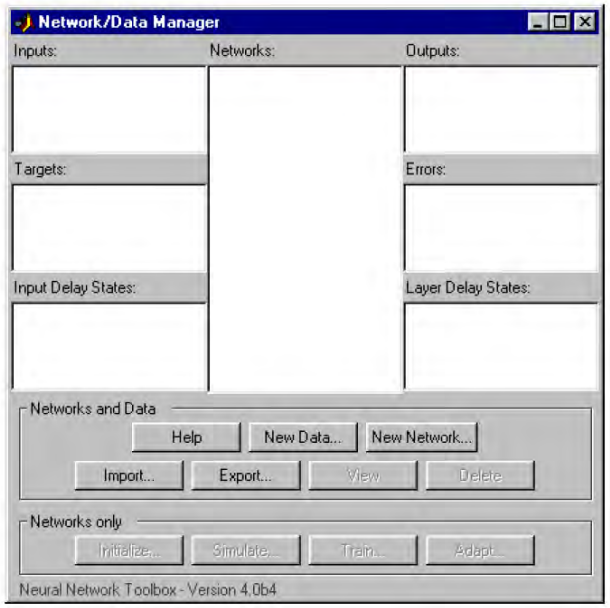
U.S. Patent No. RE49,461	nnet

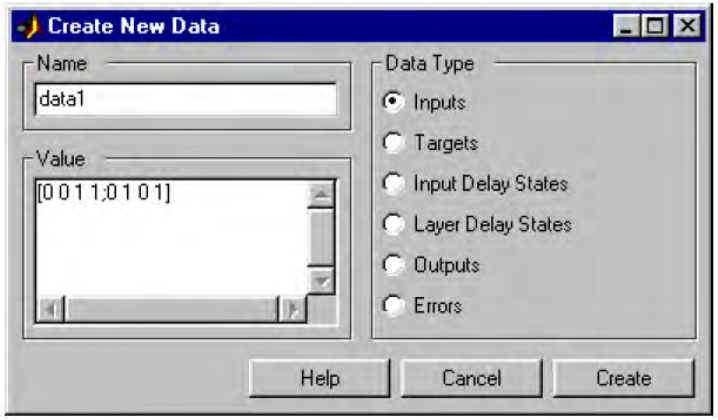
	
	Enter the following command to see how `mytf` is implemented.
	`type mytopf`
	You can use `mytopf` as a template to create your own topology function.
	Distance Functions
	Distance functions calculate the distances of a layer's neurons given their position.
	Once defined, you can assign your distance function to any layer of a network. For example, the following line of code assigns the topology function `yourdistf` to the second layer of a network.
	`net.layers{2}.distanceFcn = 'yourdistf';`
	Your distance function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	412.

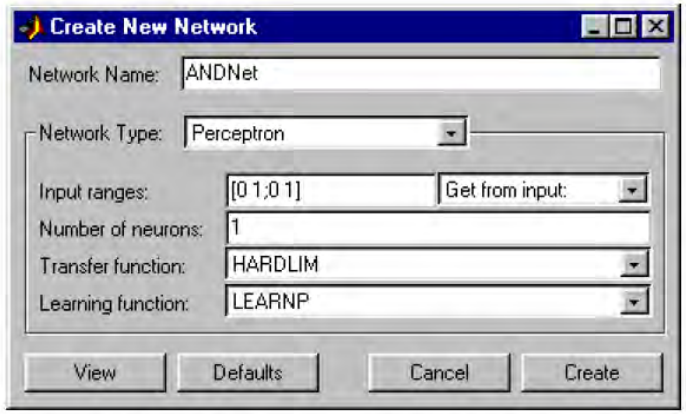
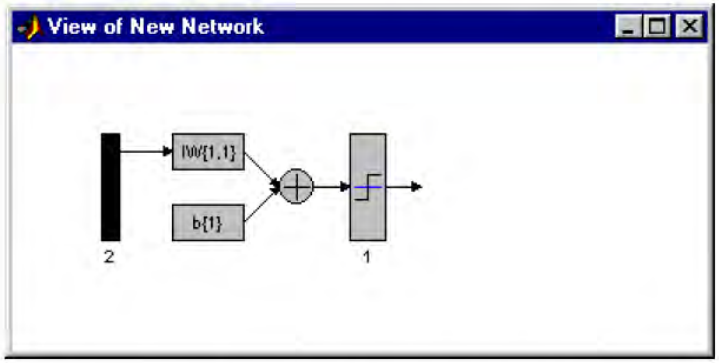
U.S. Patent No. RE49,461	nnet

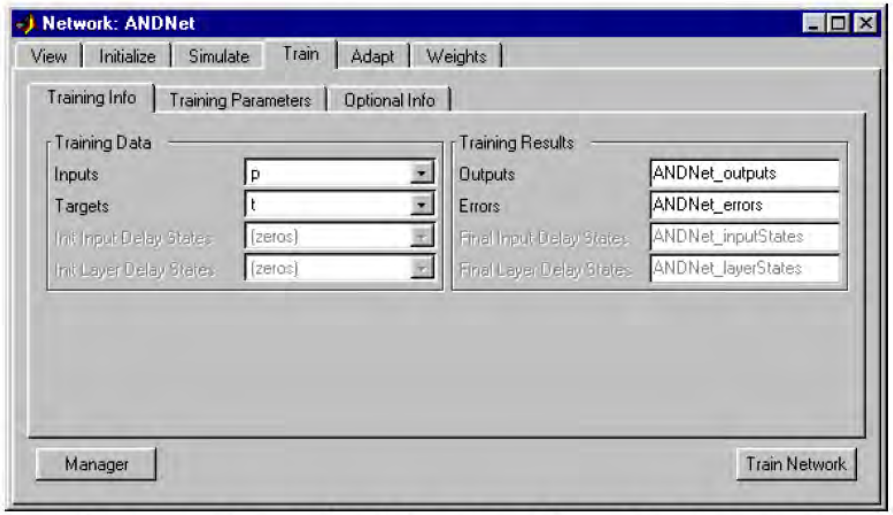
	To be a valid distance function, it must calculate distances `d` from position `pos` as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• `pos` is an $N \times S$ matrix of S neuron position vectors.
	• `d` is an $S \times S$ matrix of neuron distances.
	The toolbox contains an example custom distance function called `mydistf`. Enter the following lines of code to see how it is used.
	`help mydistf`
	`pos = gridtop(4,5);`
	`d = mydistf(pos)`
	Enter the following command to see how `mytf` is implemented.
	`type mydistf`
	You can use `mydistf` as a template to create your own distance function.
	
	413.
	
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.

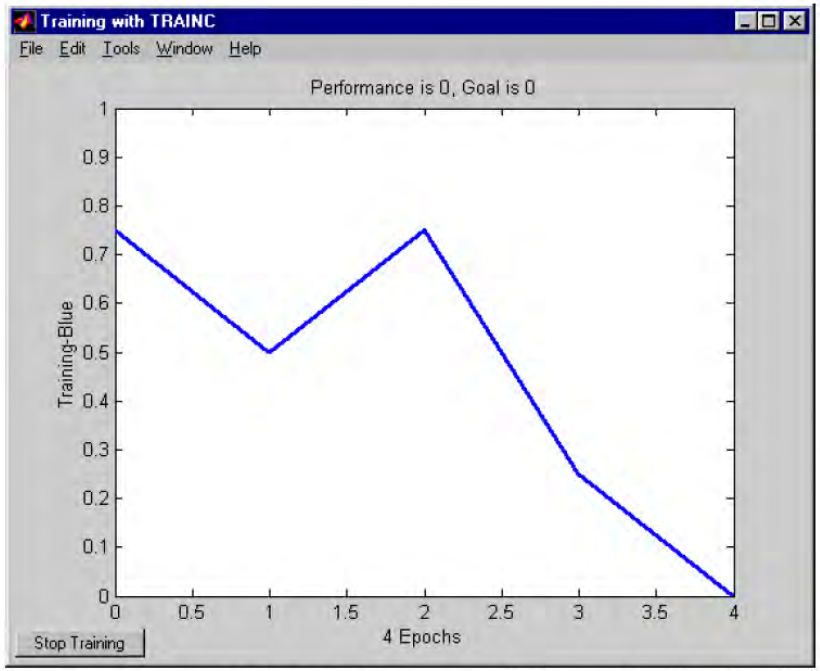
U.S. Patent No. RE49,461	nnet
24. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the computations representing the artificial neural network.	nnet discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of parameters of the computations representing the artificial neural network. *See* claims 21, 23; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses controlling setting and editing of parameters of the computations representing the artificial neural network (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

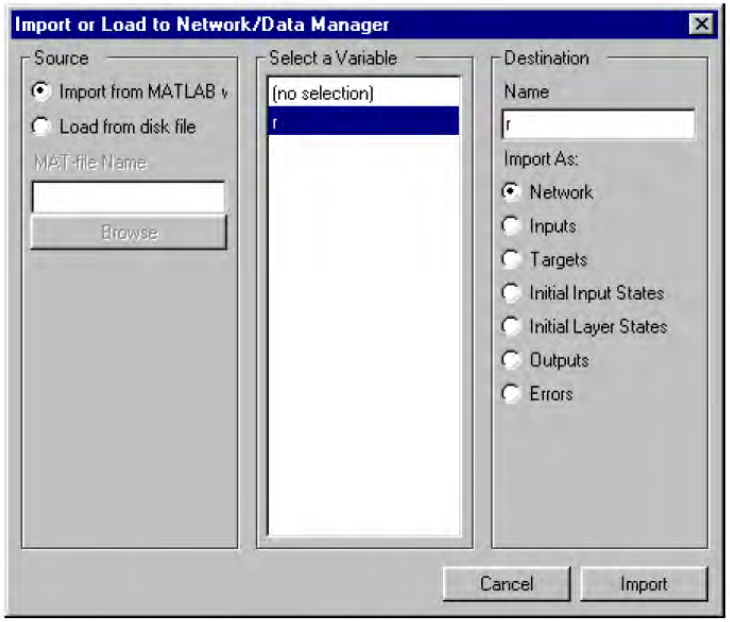
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

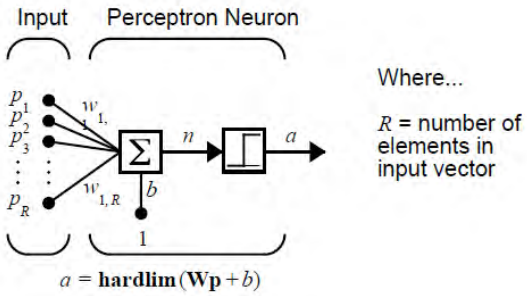
U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

U.S. Patent No. RE49,461	nnet
	392.

12 Advanced Topics

Custom Functions

The toolbox allows you to create and use many kinds of functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for your networks.

The following sections describe how to create your own versions of these kinds of functions:

- Simulation functions
  - transfer functions
  - net input functions
  - weight functions
- Initialization functions
  - network initialization functions
  - layer initialization functions
  - weight and bias initialization functions
- Learning functions
  - network training functions
  - network adapt functions
  - network performance functions
  - weight and bias learning functions
- Self-organizing map functions
  - topology functions
  - distance functions

Simulation Functions

You can create three kinds of simulation functions: transfer, net input, and weight functions. You can also provide associated derivative functions to enable backpropagation learning with your functions.

Transfer Functions

Transfer functions calculate a layer's output vector (or matrix) A, given its net input vector (or matrix) N. The only constraint on the relationship between the

U.S. Patent No. RE49,461	nnet

	output and net input is that the output must have the same dimensions as the input.
	
	Once defined, you can assign your transfer function to any layer of a network. For example, the following line of code assigns the transfer function `yourtf` to the second layer of a network.
	
	`net.layers{2}.transferFcn = 'yourtf';`
	
	Your transfer function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid transfer function, your function must calculate outputs `A` from net inputs `N` as follows,
	
	`A = yourtf(N)`
	
	where:
	
	• `N` is an S x Q matrix of Q net input (column) vectors.
	• `A` is an S x Q matrix of Q output (column) vectors.
	
	Your transfer function must also provide information about itself, using this calling format,
	
	`info = yourtf(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'output'` - Returns the output range.
	• `'active'` - Returns the active input range.
	
	The toolbox contains an example custom transfer function called `mytf`. Enter the following lines of code to see how it is used.
	
	`help mytf`
	`n = -5:.1:5;`
	`a = mytf(n);`
	`plot(n,a)`
	`mytf('deriv')`
	
	393.

U.S. Patent No. RE49,461	nnet

	Enter the following command to see how `mytf` is implemented.
	`type mytf`
	You can use `mytf` as a template to create your own transfer function.
	Transfer Derivative Functions. If you want to use backpropagation with your custom transfer function, you need to create a custom derivative function for it. The function needs to calculate the derivative of the layer's output with respect to its net input,
	`dA_dN = yourdtf(N,A)`
	where:
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q output (column) vectors.
	• `dA_dN` is the $S \times Q$ derivative dA/dN.
	This only works for transfer functions whose output elements are independent. In other words, where each `A(i)` is only a function of `N(i)`. Otherwise, a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom transfer derivative function `mydtf` works, type
	`help mydtf`
	`da_dn = mydtf(n,a)`
	`subplot(2,1,1), plot(n,a)`
	`subplot(2,1,2), plot(n,dn_da)`
	Use this command to see how `mydtf` was implemented.
	`type mydtf`
	You can use `mydtf` as a template to create your own transfer derivative functions.
	Net Input Functions
	Net input functions calculate a layer's net input vector (or matrix) `N`, given its weighted input vectors (or matrices) `Zi`. The only constraints on the relationship between the net input and the weighted inputs are that the net
	394.

U.S. Patent No. RE49,461	nnet

	input must have the same dimensions as the weighted inputs, and that the function cannot be sensitive to the order of the weight inputs.
	
	Once defined, you can assign your net input function to any layer of a network. For example, the following line of code assigns the transfer function `yournif` to the second layer of a network.
	
	`net.layers{2}.netInputFcn = 'yournif';`
	
	Your net input function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid net input function your function must calculate outputs `A` from net inputs `N` as follows,
	
	`N = yournif(Z1,Z2,...)`
	
	where
	
	• `Zi` is the ith $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	
	Your net input function must also provide information about itself using this calling format,
	
	`info = yournif(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	
	The toolbox contains an example custom net input function called `mynif`. Enter the following lines of code to see how it is used.
	
	`help mynif`
	`z1 = rand(4,5);`
	`z2 = rand(4,5);`
	`z3 = rand(4,5);`
	`n = mynif(z1,z2,z3)`
	`mynif('deriv')`
	
	Enter the following command to see how `mynif` is implemented.
	
	`type mynif`
	
	395.

U.S. Patent No. RE49,461	nnet

	You can use `mynif` as a template to create your own net input function.
	
	Net Input Derivative Functions. If you want to use backpropagation with your custom net input function, you need to create a custom derivative function for it. It needs to calculate the derivative of the layer's net input with respect to any of its weighted inputs,
	`dN_dZ = dtansig(Z,N)`
	where:
	• `Z` is one of the $S \times Q$ matrices of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `dN_dZ` is the $S \times Q$ derivative dN/dZ.
	To see how the example custom net input derivative function `mydtf` works, type
	`help mydnif`
	`dn_dz1 = mydnif(z1,n)`
	`dn_dz2 = mydnif(z1,n)`
	`dn_dz3 = mydnif(z1,n)`
	Use this command to see how `mydtf` was implemented.
	`type mydnif`
	You can use `mydnif` as a template to create your own net input derivative functions.
	
	Weight Functions
	Weight functions calculate a weighted input vector (or matrix) `Z`, given an input vector (or matrices) `P` and a weight matrix `W`.
	Once defined, you can assign your weight function to any input weight or layer weight of a network. For example, the following line of code assigns the weight function `yourwf` to the weight going to the second layer from the first input of a network.
	`net.inputWeights{2,1}.weightFcn = 'yourwf';`
	Your weight function is used whenever you simulate your network.
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	396.

U.S. Patent No. RE49,461	nnet

	To be a valid weight function your function must calculate weight inputs Z from inputs P and a weight matrix W as follows,
	`Z = yourwf(W,P)`
	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	Your net input function must also provide information about itself using this calling format,
	`info = yourwf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	The toolbox contains an example custom weight called mywf. Enter the following lines of code to see how it is used.
	`help mywf`
	`w = rand(1,5);`
	`p = rand(5,1);`
	`z = mywf(w,p);`
	`mywf('deriv')`
	Enter the following command to see how mywf is implemented.
	`type mywf`
	You can use mywf as a template to create your own weight functions.
	Weight Derivative Functions. If you want to use backpropagation with your custom weight function, you need to create a custom derivative function for it. It needs to calculate the derivative of the weight inputs with respect to both the input and weight,
	`dZ_dP = mydwf('p',W,P,Z)`
	`dZ_dW = mydwf('w',W,P,Z)`
	
	397.

U.S. Patent No. RE49,461	nnet

	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• dZ_dP is the $S \times R$ derivative dZ/dP.
	• dZ_dW is the $R \times Q$ derivative dZ/dW.
	This only works for weight functions whose output consists of a sum of i term, where each ith term is a function of only W(i) and P(i). Otherwise a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom net input derivative function mydwf works, type
	`help mydwf`
	`dz_dp = mydwf('p',w,p,z)`
	`dz_dw = mydwf('w',w,p,z)`
	Use this command to see how mydwf is implemented.
	`type mydwf`
	You can use mydwf as a template to create your own net input derivative function.
	Initialization Functions
	You can create three kinds of initialization functions: network, layer, and weight/bias initialization.
	Network Initialization Functions
	The most general kind of initialization function is the network initialization function which sets all the weights and biases of a network to values suitable as a starting point for training or adaption.
	Once defined, you can assign your network initialization function to a network.
	`net.initFcn = 'yournif';`
	398.

U.S. Patent No. RE49,461	nnet

	Your network initialization function is used whenever you initialize your network.
	`net = init(net)`
	To be a valid network initialization function, it must take and return a network.
	`net = yournif(net)`
	Your function can set the network's weight and bias values in any way you want. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `init` turns off the normal type checking for network properties before calling your initialization function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but could cause problems later when you try to simulate or train the network.
	You can examine the implementation of the toolbox function `initlay` if you are interested in creating your own network initialization function.
	Layer Initialization Functions
	The layer initialization function sets all the weights and biases of a layer to values suitable as a starting point for training or adaption.
	Once defined, you can assign your layer initialization function to a layer of a network. For example, the following line of code assigns the layer initialization function `yourlif` to the second layer of a network.
	`net.layers{2}.initFcn = 'yourlif';`
	Layer initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`. If this is the case, then your function is used to initialize the layer whenever you initialize your network with `init`.
	`net = init(net)`
	To be a valid layer initialization function, it must take a network and a layer index `i`, and return the network after initializing the *i*th layer.
	`net = yournif(net,i)`
	399.

U.S. Patent No. RE49,461	nnet

	Your function can then set the *i*th layer's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size.
	
	If you are interested in creating your own layer initialization function, you can examine the implementations of the toolbox functions `initwb` and `initnw`.
	
	Weight and Bias Initialization Functions
	The weight and bias initialization function sets all the weights and biases of a weight or bias to values suitable as a starting point for training or adaption.
	
	Once defined, you can assign your initialization function to any weight and bias in a network. For example, the following lines of code assign the weight and bias initialization function `yourwbif` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.initFcn = 'yourwbif';`
	`net.inputWeights{2,1}.initFcn = 'yourwbif';`
	
	Weight and bias initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`, and the layer's initialization function (`net.layers{i}.initFcn`) is set to the toolbox function `initwb`. If this is the case, then your function is used to initialize the weight and biases it is assigned to whenever you initialize your network with `init`.
	
	`net = init(net)`
	
	To be a valid weight and bias initialization function, it must take a the number of neurons in a layer `S`, and a two-column matrix `PR` of R rows defining the minimum and maximum values of R inputs and return a new weight matrix `W`,
	
	`W = rands(S,PR)`
	
	where:
	
	• `S` is the number of neurons in the layer.
	• `PR` is an $R \times 2$ matrix defining the minimum and maximum values of R inputs.
	• `W` is a new $S \times R$ weight matrix.
	
	Your function also needs to generate a new bias vector as follows,
	
	400.

U.S. Patent No. RE49,461	nnet
	b = rands(S) where: • S is the number of neurons in the layer. • b is a new $S \times 1$ bias vector. To see how an example custom weight and bias initialization function works, type help mywbif W = mywbif(4,[0 1; -2 2]) b = mywbif(4,[1 1]) Use this command to see how mywbif was implemented. type mywbif You can use mywbif as a template to create your own weight and bias initialization function. **Learning Functions** You can create four kinds of initialization functions: training, adaption, performance, and weight/bias learning. **Training Functions** One kind of general learning function is a network training function. Training functions repeatedly apply a set of input vectors to a network, updating the network each time, until some stopping criteria is met. Stopping criteria can consists of a maximum number of epochs, a minimum error gradient, an error goal, etc. Once defined, you can assign your training function to a network. net.trainFcn = 'yourtf'; Your network initialization function is used whenever you train your network. [net,tr] = train(NET,P,T,Pi,Ai) To be a valid training function your function must take and return a network. [net,tr] = yourtf(net,Pd,Tl,Ai,Q,TS,VV,TV) 401.

U.S. Patent No. RE49,461	nnet
	where: • Pd is an $N_l \times N_i \times$ TS cell array of tap delayed inputs. ▪ Each Pd{i,j,ts} is the $R^j \times (D_i^{ij}Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step ts. (Pd{i,j,ts} is an empty matrix [] if the ith layer doesn't have a weight from the jth input.) • Tl is an $N_l \times$ TS cell array of layer targets. ▪ Each Tl{i,ts} is the $S^i \times Q$ target matrix for the ith layer. (Tl{i,ts} is an empty matrix if the ith layer doesn't have a target.) • Ai is an $N_l \times LD$ cell array of initial layer delay states. ▪ Each Ai{1,k} is the $S^i \times Q$ delayed ith layer output for time step ts = k-LD, where ts goes from 0 to LD-1. • Q is the number of concurrent vectors. • TS is the number of time steps. • VV and TV are optional structures defining validation and test vectors in the same form as the training vectors defined above: Pd, Tl, Ai, Q, and TS. Note that the validation and testing Q and TS values can be different from each other and from those used by the training vectors. The dimensions above have the following definitions: • N_l is the number of network layers (net.numLayers). • N_i is the number of network inputs (net.numInputs). • R^j is the size of the jth input (net.inputs{j}.size). • S^i is the size of the ith layer (net.layers{i}.size) • LD is the number of layer delays (net.numLayerDelays). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (length(net.inputWeights{i,j}.delays)). Your training function must also provide information about itself using this calling format, `info = yourtf(code)` where the correct information is returned for each of the following string codes: • 'version' - Returns the Neural Network Toolbox version (3.0). • 'pdefaults' - Returns a structure of default training parameters. 402.

U.S. Patent No. RE49,461	nnet
	When you set the network training function (`net.trainFcn`) to be your function, the network's training parameters (`net.trainParam`) automatically is set to your default structure. Those values can be altered (or not) before training. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size. For performance reasons, `train` turns off the normal type checking for network properties before calling your training function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or adapt the network. If you are interested in creating your own training function, you can examine the implementations of toolbox functions such as `trainc` and `trainr`. The help for each of these utility functions lists the input and output arguments they take. **Utility Functions.** If you examine training functions such as `trainc`, `traingd`, and `trainlm`, note that they use a set of utility functions found in the `nnet/nnutils` directory. These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you might have to update your functions for future versions of the toolbox. Use `help` on each function to view the function's input and output arguments. These two functions are useful for creating a new training record and truncating it once the final number of epochs is known: • `newtr` - New training record with any number of optional fields. • `cliptr` - Clip training record to the final number of epochs. These three functions calculate network signals going forward, errors, and derivatives of performance coming back: • `calca` - Calculate network outputs and other signals. • `calcerr` - Calculate matrix or cell array errors. • `calcgrad` - Calculate bias and weight performance gradients. 403.

U.S. Patent No. RE49,461	nnet

	These two functions get and set a network's weight and bias values with single vectors. Being able to treat all these adjustable parameters as a single vector is often useful for implementing optimization algorithms:
	• `getx` - Get all network weight and bias values as a single vector.
	• `setx` - Set all network weight and bias values with a single vector.
	These next three functions are also useful for implementing optimization functions. One calculates all network signals going forward, including errors and performance. One backpropagates to find the derivatives of performance as a single vector. The third function backpropagates to find the Jacobian of performance. This latter function is used by advanced optimization techniques like Levenberg-Marquardt:
	• `calcperf` - Calculate network outputs, signals, and performance.
	• `calcgx` - Calculate weight and bias performance gradient as a single vector.
	• `calcjx` - Calculate weight and bias performance Jacobian as a single matrix.
	Adapt Functions
	The other kind of the general learning function is a network adapt function. Adapt functions simulate a network, while updating the network for each time step of the input before continuing the simulation to the next input.
	Once defined, you can assign your adapt function to a network.
	`net.adaptFcn = 'youraf';`
	Your network initialization function is used whenever you adapt your network.
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid adapt function, it must take and return a network,
	`[net,Ac,El] = youraf(net,Pd,Tl,Ai,Q,TS)`
	where:
	• `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
	- Each `Pd{i,j,ts}` is the $R^j \times (D_i{}^{ij}Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step `ts`. Note that (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.)
	404.

U.S. Patent No. RE49,461	nnet
	• `Tl` is an $N_l \times TS$ cell array of layer targets. - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. Note that (`Tl{i,ts}` is an empty matrix if the ith layer doesn't have a target.) • `Ai` is an $N_l \times LD$ cell array of initial layer delay states. - Each `Ai{l,k}` is the $S^i \times Q$ delayed ith layer output for time step `ts` = `k`-LD, where ts goes from 0 to LD-1. • `Q` is the number of concurrent vectors. • `TS` is the number of time steps. The dimensions above have the following definitions: • N_l is the number of network layers (`net.numLayers`). • N_i is the number of network inputs (`net.numInputs`). • R^j is the size of the jth input (`net.inputs{j}.size`). • S^i is the size of the ith layer (`net.layers{i}.size`) • LD is the number of layer delays (`net.numLayerDelays`). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (`length(net.inputWeights{i,j}.delays)`). Your adapt function must also provide information about itself using this calling format, `info = youraf(code)` where the correct information is returned for each of the following string codes: • `'version'` - Returns the Neural Network Toolbox version (3.0). • `'pdefaults'` - Returns a structure of default adapt parameters. When you set the network adapt function (`net.adaptFcn`) to be your function, the network's adapt parameters (`net.adaptParam`) automatically is set to your default structure. Those values can then be altered (or not) before adapting. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `adapt` turns off the normal type checking for network properties before calling your adapt function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or train the network. 405.

U.S. Patent No. RE49,461	nnet

	If you are interested in creating your own training function, you can examine the implementation of a toolbox function such as `trains`.
	
	Utility Functions. If you examine the toolbox's only adapt function `trains`, note that it uses a set of utility functions found in the `nnet/nnutils` directory. The help for each of these utility functions lists the input and output arguments they take.
	
	These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you will have to update your functions for future versions of the toolbox.
	
	These two functions are useful for simulating a network, and calculating its derivatives of performance:
	
	• `calca1` - New training record with any number of optional fields.
	• `calce1` - Clip training record to the final number of epochs.
	• `calcgrad` - Calculate bias and weight performance gradients.
	
	Performance Functions
	Performance functions allow a network's behavior to be graded. This is useful for many algorithms, such as backpropagation, which operate by adjusting network weights and biases to improve performance.
	
	Once defined you can assign your training function to a network.
	
	`net.performFcn = 'yourpf';`
	
	Your network initialization function will then be used whenever you train your adapt your network.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid performance function your function must be called as follows,
	
	`perf = yourpf(E,X,PP)`
	
	where:
	
	• `E` is either an S x Q matrix or an $N_l \times TS$ cell array of layer errors.
	
	406.

U.S. Patent No. RE49,461	nnet

	- Each `E{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl(i,ts)` is an empty matrix if the ith layer doesn't have a target.)
	• `X` is an M x 1 vector of all the network's weights and biases.
	• `PP` is a structure of network performance parameters.
	If E is a cell array you can convert it to a matrix as follows.
	`E = cell2mat(E);`
	Alternatively, your function must also be able to be called as follows,
	`perf = yourpf(E,net)`
	where you can get `X` and `PP` (if needed) as follows.
	`X = getx(net);`
	`PP = net.performParam;`
	Your performance function must also provide information about itself using this calling format,
	`info = yourpf(code)`
	where the correct information is returned for each of the following string codes:
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'pdefaults'` - Returns a structure of default performance parameters.
	When you set the network performance function (`net.performFcn`) to be your function, the network's adapt parameters (`net.performParam`) will automatically get set to your default structure. Those values can then be altered or not before training or adaption.
	To see how an example custom performance function works type in these lines of code.
	`help mypf`
	`e = rand(4,5);`
	`x = rand(12,1);`
	`pp = mypf('pdefaults')`
	`perf = mypf(e,x,pp)`
	
	407.

U.S. Patent No. RE49,461	nnet

	Use this command to see how mypf was implemented.
	`type mypf`
	You can use mypf as a template to create your own weight and bias initialization function.
	Performance Derivative Functions. If you want to use backpropagation with your performance function, you need to create a custom derivative function for it. It needs to calculate the derivative of the network's errors and combined weight and bias vector, with respect to performance,
	`dPerf_dE = dmsereg('e',E,X,perf,PP)`
	`dPerf_dX = dmsereg('x',E,X,perf,PP)`
	where:
	• E is an $N_l \times TS$ cell array of layer errors.
	- Each E{i,ts} is the $S^i \times Q$ target matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• X is an $M \times 1$ vector of all the network's weights and biases.
	• PP is a structure of network performance parameters.
	• dPerf_dE is the $N_l \times TS$ cell array of derivatives dPerf/dE.
	- Each E{i,ts} is the $S^i \times Q$ derivative matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• dPerf_dX is the $M \times 1$ derivative dPerf/dX.
	To see how the example custom performance derivative function mydpf works, type
	`help mydpf`
	`e = {e};`
	`dperf_de = mydpf('e',e,x,perf,pp)`
	`dperf_dx = mydpf('x',e,x,perf,pp)`
	Use this command to see how mydpf was implemented.
	`type mydpf`
	You can use mydpf as a template to create your own performance derivative functions.
	408.

U.S. Patent No. RE49,461	nnet

	Weight and Bias Learning Functions
	The most specific kind of learning function is a weight and bias learning function. These functions are used to update individual weights and biases during learning. with some training and adapt functions.
	
	Once defined. you can assign your learning function to any weight and bias in a network. For example, the following lines of code assign the weight and bias learning function `yourwblf` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.learnFcn = 'yourwblf';`
	`net.inputWeights{2,1}.learnFcn = 'yourwblf';`
	
	Weight and bias learning functions are only called to update weights and biases if the network training function (`net.trainFcn`) is set to `trainb`, `trainc`, or `trainr`, or if the network adapt function (`net.adaptFcn`) is set to `trains`. If this is the case, then your function is used to update the weight and biases it is assigned to whenever you train or adapt your network with `train` or `adapt`.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid weight and bias learning function, it must be callable as follows,
	
	`[dW,LS] = yourwblf(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)`
	
	where:
	
	• `W` is an $S \times R$ weight matrix.
	• `P` is an $R \times Q$ matrix of Q input (column) vectors.
	• `Z` is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q layer output (column) vectors.
	• `T` is an $S \times Q$ matrix of Q target (column) vectors.
	• `E` is an $S \times Q$ matrix of Q error (column) vectors.
	• `gW` is an $S \times R$ gradient of `W` with respect to performance.
	• `gA` is an $S \times Q$ gradient of `A` with respect to performance.
	• `D` is an $S \times S$ matrix of neuron distances.
	• `LP` is a a structure of learning parameters.
	
	409.

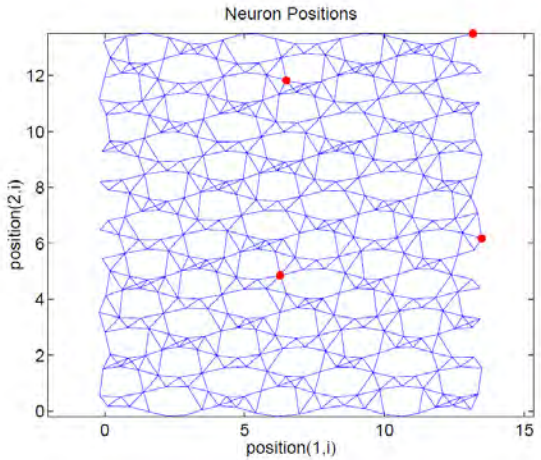
U.S. Patent No. RE49,461	nnet

	• LS is a structure of the learning state that is updated for each call. (Use a null matrix [ ] the first time.)
	• dW is the resulting $S \times R$ weight change matrix.
	Your function is called as follows to update bias vector
	`[db,LS] = yourwblf(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)`
	where:
	• S is the number of neurons in the layer.
	• b is a new $S \times 1$ bias vector.
	Your learning function must also provide information about itself using this calling format,
	`info = yourwblf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	• 'pdefaults' - Returns a structure of default performance parameters.
	To see how an example custom weight and bias initialization function works, type
	`help mywblf`
	Use this command to see how mywbif was implemented.
	`type mywblf`
	You can use mywblf as a template to create your own weight and bias learning function.
	Self-Organizing Map Functions
	There are two kinds of functions that control how neurons in self-organizing maps respond. They are topology and distance functions.
	Topology Functions
	Topology functions calculate the positions of a layer's neurons given its dimensions.
	
	410.

U.S. Patent No. RE49,461	nnet

	Once defined, you can assign your topology function to any layer of a network. For example, the following line of code assigns the topology function `yourtopf` to the second layer of a network.
	`net.layers{2}.topologyFcn = 'yourtopf';`
	Your topology function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid topology function your function must calculate positions `pos` from dimensions `dim` as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• `dimi` is the number of neurons along the ith dimension of the layer.
	• `pos` is an $N \times S$ matrix of S position vectors, where S is the total number of neurons that is defined by the product `dim1*dim1*...*dimN`.
	The toolbox contains an example custom topology function called `mytopf`. Enter the following lines of code to see how it is used.
	`help mytopf`
	`pos = mytopf(20,20);`
	`plotsom(pos)`
	If you type that code, you get the following plot.
	
	411.

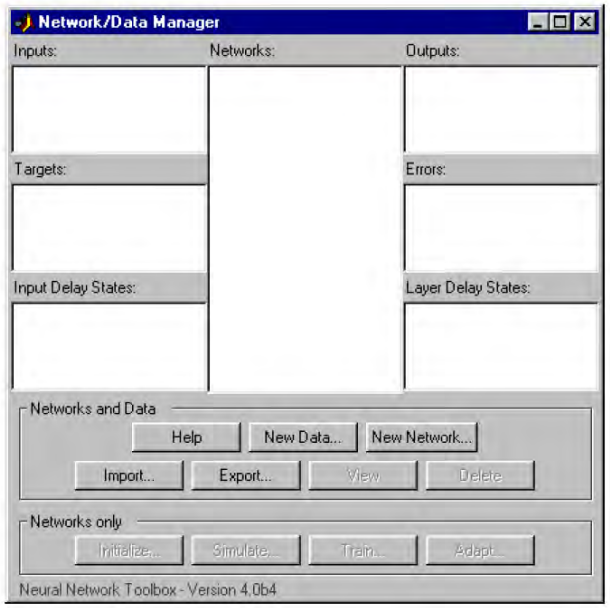
U.S. Patent No. RE49,461	nnet

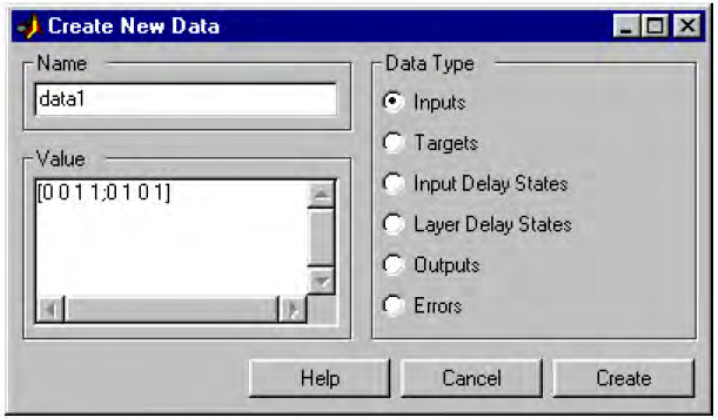
	
	Enter the following command to see how mytf is implemented.
	`type mytopf`
	You can use mytopf as a template to create your own topology function.
	Distance Functions
	Distance functions calculate the distances of a layer's neurons given their position.
	Once defined, you can assign your distance function to any layer of a network. For example, the following line of code assigns the topology function yourdistf to the second layer of a network.
	`net.layers{2}.distanceFcn = 'yourdistf';`
	Your distance function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	412.

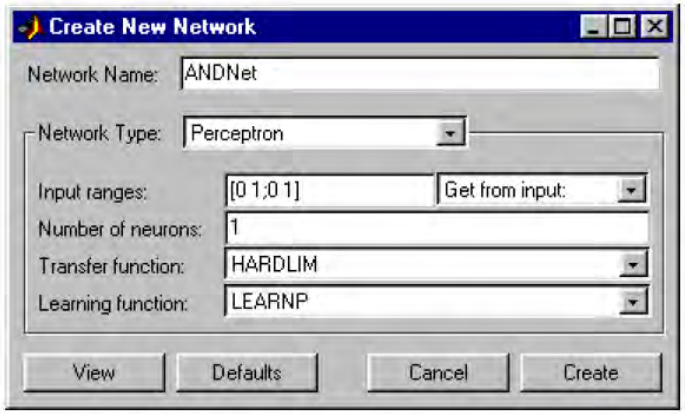
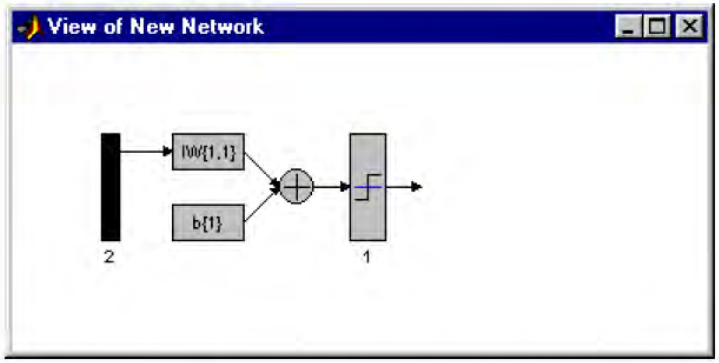
U.S. Patent No. RE49,461	nnet

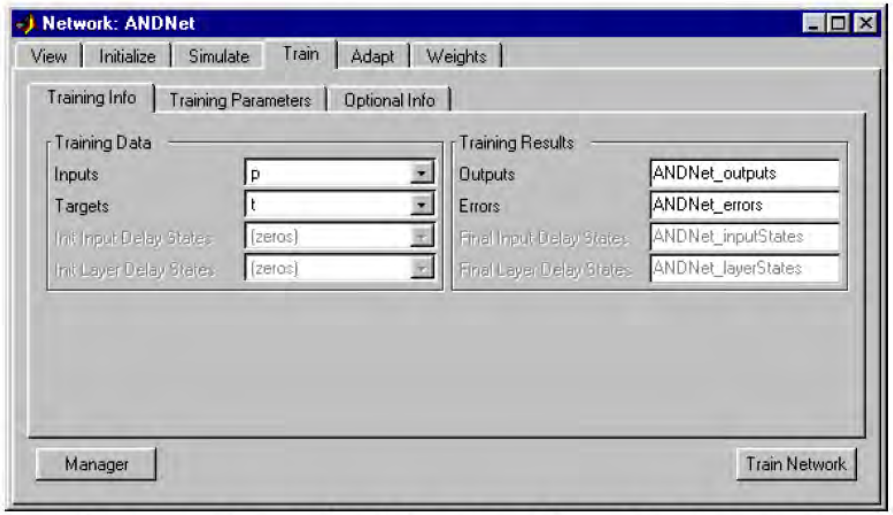
	To be a valid distance function, it must calculate distances `d` from position `pos` as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• `pos` is an $N \times S$ matrix of S neuron position vectors.
	• `d` is an $S \times S$ matrix of neuron distances.
	The toolbox contains an example custom distance function called `mydistf`. Enter the following lines of code to see how it is used.
	`help mydistf`
	`pos = gridtop(4,5);`
	`d = mydistf(pos)`
	Enter the following command to see how `mytf` is implemented.
	`type mydistf`
	You can use `mydistf` as a template to create your own distance function.
	
	413.
	
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.

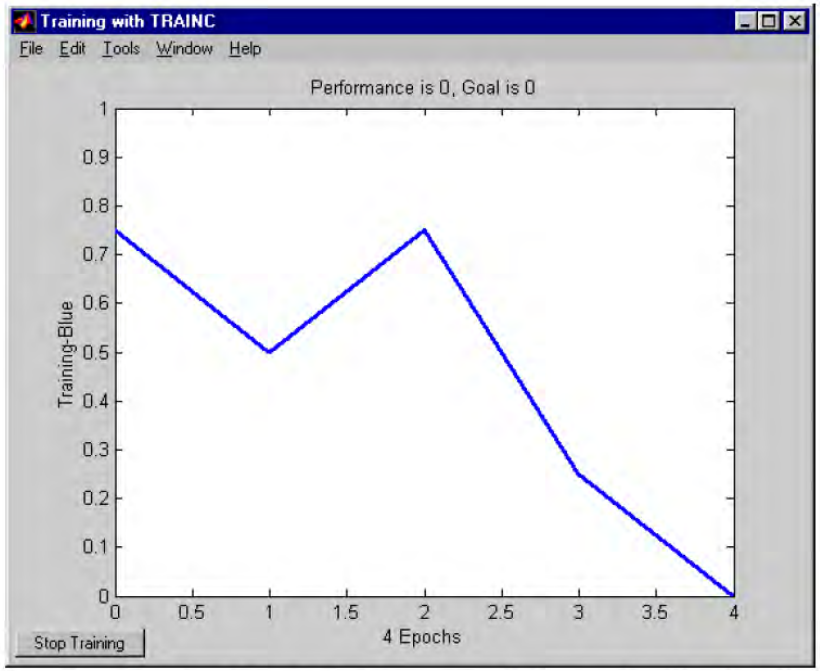
U.S. Patent No. RE49,461	nnet
25. The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the inputs to the artificial neural network.	nnet discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises controlling setting and editing of parameters of the inputs to the artificial neural network. *See* claim 21; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses controlling setting and editing of parameters of the inputs to the artificial network (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
   2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
   ans =
   -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

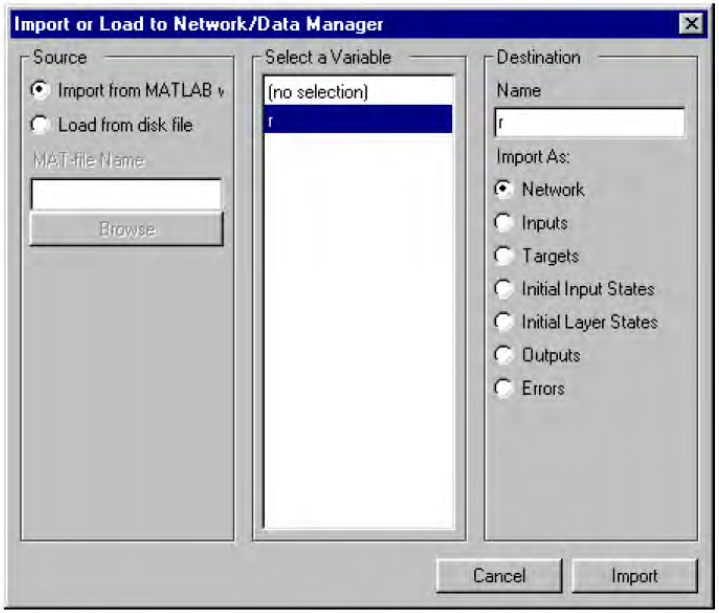
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

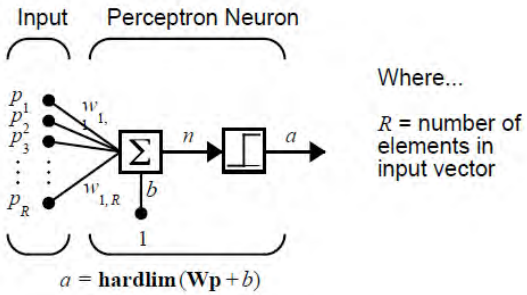
Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

U.S. Patent No. RE49,461	nnet
	What's New in Version 4.0 A few of the new features and improvements introduced with this version of the Neural Network Toolbox are discussed below. **Control System Applications** A new Chapter 6 presents three practical control systems applications: • Network model predictive control • Model reference adaptive control • Feedback linearization controller **Graphical User Interface** A graphical user interface has been added to the toolbox. This interface allows you to: • Create networks • Enter data into the GUI • Initialize, train, and simulate networks • Export the training results from the GUI to the command line workspace • Import data from the command line workspace to the GUI To open the Network/Data Manager window type `nntool`. **New Training Functions** The toolbox now has four training algorithms that apply weight and bias learning rules. One algorithm applies the learning rules in batch mode. Three algorithms apply learning rules in three different incremental modes: • `trainb` - Batch training function • `trainc` - Cyclical order incremental training function • `trainr` - Random order incremental training function • `trains` - Sequential order incremental training function All four functions present the whole training set in each epoch (pass through the entire input set). 31.

U.S. Patent No. RE49,461	nnet
	392.

12 Advanced Topics

Custom Functions

The toolbox allows you to create and use many kinds of functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for your networks.

The following sections describe how to create your own versions of these kinds of functions:

- Simulation functions
  - transfer functions
  - net input functions
  - weight functions
- Initialization functions
  - network initialization functions
  - layer initialization functions
  - weight and bias initialization functions
- Learning functions
  - network training functions
  - network adapt functions
  - network performance functions
  - weight and bias learning functions
- Self-organizing map functions
  - topology functions
  - distance functions

Simulation Functions

You can create three kinds of simulation functions: transfer, net input, and weight functions. You can also provide associated derivative functions to enable backpropagation learning with your functions.

Transfer Functions

Transfer functions calculate a layer's output vector (or matrix) A, given its net input vector (or matrix) N. The only constraint on the relationship between the

U.S. Patent No. RE49,461	nnet

	output and net input is that the output must have the same dimensions as the input.
	
	Once defined, you can assign your transfer function to any layer of a network. For example, the following line of code assigns the transfer function `yourtf` to the second layer of a network.
	
	`net.layers{2}.transferFcn = 'yourtf';`
	
	Your transfer function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid transfer function, your function must calculate outputs `A` from net inputs `N` as follows,
	
	`A = yourtf(N)`
	
	where:
	
	• `N` is an S x Q matrix of Q net input (column) vectors.
	• `A` is an S x Q matrix of Q output (column) vectors.
	
	Your transfer function must also provide information about itself, using this calling format,
	
	`info = yourtf(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'output'` - Returns the output range.
	• `'active'` - Returns the active input range.
	
	The toolbox contains an example custom transfer function called `mytf`. Enter the following lines of code to see how it is used.
	
	`help mytf`
	`n = -5:.1:5;`
	`a = mytf(n);`
	`plot(n,a)`
	`mytf('deriv')`
	
	393.

U.S. Patent No. RE49,461	nnet

	Enter the following command to see how `mytf` is implemented.
	`type mytf`
	You can use `mytf` as a template to create your own transfer function.
	Transfer Derivative Functions. If you want to use backpropagation with your custom transfer function, you need to create a custom derivative function for it. The function needs to calculate the derivative of the layer's output with respect to its net input,
	`dA_dN = yourdtf(N,A)`
	where:
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q output (column) vectors.
	• `dA_dN` is the $S \times Q$ derivative dA/dN.
	This only works for transfer functions whose output elements are independent. In other words, where each `A(i)` is only a function of `N(i)`. Otherwise, a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom transfer derivative function `mydtf` works, type
	`help mydtf`
	`da_dn = mydtf(n,a)`
	`subplot(2,1,1), plot(n,a)`
	`subplot(2,1,2), plot(n,dn_da)`
	Use this command to see how `mydtf` was implemented.
	`type mydtf`
	You can use `mydtf` as a template to create your own transfer derivative functions.
	Net Input Functions
	Net input functions calculate a layer's net input vector (or matrix) `N`, given its weighted input vectors (or matrices) `Zi`. The only constraints on the relationship between the net input and the weighted inputs are that the net
	394.

U.S. Patent No. RE49,461	nnet

	input must have the same dimensions as the weighted inputs, and that the function cannot be sensitive to the order of the weight inputs.
	
	Once defined, you can assign your net input function to any layer of a network. For example, the following line of code assigns the transfer function `yournif` to the second layer of a network.
	
	`net.layers{2}.netInputFcn = 'yournif';`
	
	Your net input function then is used whenever you simulate your network.
	
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	To be a valid net input function your function must calculate outputs `A` from net inputs `N` as follows,
	
	`N = yournif(Z1,Z2,...)`
	
	where
	
	• `Zi` is the ith $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	
	Your net input function must also provide information about itself using this calling format,
	
	`info = yournif(code)`
	
	where the correct information is returned for each of the following string codes:
	
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	
	The toolbox contains an example custom net input function called `mynif`. Enter the following lines of code to see how it is used.
	
	`help mynif`
	`z1 = rand(4,5);`
	`z2 = rand(4,5);`
	`z3 = rand(4,5);`
	`n = mynif(z1,z2,z3)`
	`mynif('deriv')`
	
	Enter the following command to see how `mynif` is implemented.
	
	`type mynif`
	
	395.

U.S. Patent No. RE49,461	nnet

	You can use `mynif` as a template to create your own net input function.
	
	Net Input Derivative Functions. If you want to use backpropagation with your custom net input function, you need to create a custom derivative function for it. It needs to calculate the derivative of the layer's net input with respect to any of its weighted inputs,
	`dN_dZ = dtansig(Z,N)`
	where:
	• `Z` is one of the $S \times Q$ matrices of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `dN_dZ` is the $S \times Q$ derivative dN/dZ.
	To see how the example custom net input derivative function `mydtf` works, type
	`help mydnif`
	`dn_dz1 = mydnif(z1,n)`
	`dn_dz2 = mydnif(z1,n)`
	`dn_dz3 = mydnif(z1,n)`
	Use this command to see how `mydtf` was implemented.
	`type mydnif`
	You can use `mydnif` as a template to create your own net input derivative functions.
	
	Weight Functions
	Weight functions calculate a weighted input vector (or matrix) `Z`, given an input vector (or matrices) `P` and a weight matrix `W`.
	Once defined, you can assign your weight function to any input weight or layer weight of a network. For example, the following line of code assigns the weight function `yourwf` to the weight going to the second layer from the first input of a network.
	`net.inputWeights{2,1}.weightFcn = 'yourwf';`
	Your weight function is used whenever you simulate your network.
	`[Y,Pf,Af] = sim(net,P,Pi,Ai)`
	
	396.

U.S. Patent No. RE49,461	nnet

	To be a valid weight function your function must calculate weight inputs Z from inputs P and a weight matrix W as follows,
	`Z = yourwf(W,P)`
	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	Your net input function must also provide information about itself using this calling format,
	`info = yourwf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	The toolbox contains an example custom weight called mywf. Enter the following lines of code to see how it is used.
	`help mywf`
	`w = rand(1,5);`
	`p = rand(5,1);`
	`z = mywf(w,p);`
	`mywf('deriv')`
	Enter the following command to see how mywf is implemented.
	`type mywf`
	You can use mywf as a template to create your own weight functions.
	Weight Derivative Functions. If you want to use backpropagation with your custom weight function, you need to create a custom derivative function for it. It needs to calculate the derivative of the weight inputs with respect to both the input and weight,
	`dZ_dP = mydwf('p',W,P,Z)`
	`dZ_dW = mydwf('w',W,P,Z)`
	
	397.

U.S. Patent No. RE49,461	nnet

	where:
	• W is an $S \times R$ weight matrix.
	• P is an $R \times Q$ matrix of Q input (column) vectors.
	• Z is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• dZ_dP is the $S \times R$ derivative dZ/dP.
	• dZ_dW is the $R \times Q$ derivative dZ/dW.
	This only works for weight functions whose output consists of a sum of i term, where each ith term is a function of only W(i) and P(i). Otherwise a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.
	To see how the example custom net input derivative function mydwf works, type
	`help mydwf`
	`dz_dp = mydwf('p',w,p,z)`
	`dz_dw = mydwf('w',w,p,z)`
	Use this command to see how mydwf is implemented.
	`type mydwf`
	You can use mydwf as a template to create your own net input derivative function.
	Initialization Functions
	You can create three kinds of initialization functions: network, layer, and weight/bias initialization.
	Network Initialization Functions
	The most general kind of initialization function is the network initialization function which sets all the weights and biases of a network to values suitable as a starting point for training or adaption.
	Once defined, you can assign your network initialization function to a network.
	`net.initFcn = 'yournif';`
	398.

U.S. Patent No. RE49,461	nnet

	Your network initialization function is used whenever you initialize your network.
	`net = init(net)`
	To be a valid network initialization function, it must take and return a network.
	`net = yournif(net)`
	Your function can set the network's weight and bias values in any way you want. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `init` turns off the normal type checking for network properties before calling your initialization function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but could cause problems later when you try to simulate or train the network.
	You can examine the implementation of the toolbox function `initlay` if you are interested in creating your own network initialization function.
	Layer Initialization Functions
	The layer initialization function sets all the weights and biases of a layer to values suitable as a starting point for training or adaption.
	Once defined, you can assign your layer initialization function to a layer of a network. For example, the following line of code assigns the layer initialization function `yourlif` to the second layer of a network.
	`net.layers{2}.initFcn = 'yourlif';`
	Layer initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`. If this is the case, then your function is used to initialize the layer whenever you initialize your network with `init`.
	`net = init(net)`
	To be a valid layer initialization function, it must take a network and a layer index `i`, and return the network after initializing the *i*th layer.
	`net = yournif(net,i)`
	399.

U.S. Patent No. RE49,461	nnet

	Your function can then set the *i*th layer's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size.
	
	If you are interested in creating your own layer initialization function, you can examine the implementations of the toolbox functions `initwb` and `initnw`.
	
	Weight and Bias Initialization Functions
	The weight and bias initialization function sets all the weights and biases of a weight or bias to values suitable as a starting point for training or adaption.
	
	Once defined, you can assign your initialization function to any weight and bias in a network. For example, the following lines of code assign the weight and bias initialization function `yourwbif` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.initFcn = 'yourwbif';`
	`net.inputWeights{2,1}.initFcn = 'yourwbif';`
	
	Weight and bias initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`, and the layer's initialization function (`net.layers{i}.initFcn`) is set to the toolbox function `initwb`. If this is the case, then your function is used to initialize the weight and biases it is assigned to whenever you initialize your network with `init`.
	
	`net = init(net)`
	
	To be a valid weight and bias initialization function, it must take a the number of neurons in a layer `S`, and a two-column matrix `PR` of R rows defining the minimum and maximum values of R inputs and return a new weight matrix `W`,
	
	`W = rands(S,PR)`
	
	where:
	
	• `S` is the number of neurons in the layer.
	• `PR` is an $R \times 2$ matrix defining the minimum and maximum values of R inputs.
	• `W` is a new $S \times R$ weight matrix.
	
	Your function also needs to generate a new bias vector as follows,
	
	400.

U.S. Patent No. RE49,461	nnet
	b = rands(S) where: • S is the number of neurons in the layer. • b is a new $S \times 1$ bias vector. To see how an example custom weight and bias initialization function works, type help mywbif W = mywbif(4,[0 1; -2 2]) b = mywbif(4,[1 1]) Use this command to see how mywbif was implemented. type mywbif You can use mywbif as a template to create your own weight and bias initialization function. **Learning Functions** You can create four kinds of initialization functions: training, adaption, performance, and weight/bias learning. **Training Functions** One kind of general learning function is a network training function. Training functions repeatedly apply a set of input vectors to a network, updating the network each time, until some stopping criteria is met. Stopping criteria can consists of a maximum number of epochs, a minimum error gradient, an error goal, etc. Once defined, you can assign your training function to a network. net.trainFcn = 'yourtf'; Your network initialization function is used whenever you train your network. [net,tr] = train(NET,P,T,Pi,Ai) To be a valid training function your function must take and return a network. [net,tr] = yourtf(net,Pd,Tl,Ai,Q,TS,VV,TV) 401.

U.S. Patent No. RE49,461	nnet
	where: • Pd is an $N_l \times N_i \times TS$ cell array of tap delayed inputs. ▪ Each Pd{i,j,ts} is the $R^j \times (D_i^{ij} Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step ts. (Pd{i,j,ts} is an empty matrix [] if the ith layer doesn't have a weight from the jth input.) • Tl is an $N_l \times TS$ cell array of layer targets. ▪ Each Tl{i,ts} is the $S^i \times Q$ target matrix for the ith layer. (Tl{i,ts} is an empty matrix if the ith layer doesn't have a target.) • Ai is an $N_l \times LD$ cell array of initial layer delay states. ▪ Each Ai{l,k} is the $S^i \times Q$ delayed ith layer output for time step ts = k-LD, where ts goes from 0 to LD-1. • Q is the number of concurrent vectors. • TS is the number of time steps. • VV and TV are optional structures defining validation and test vectors in the same form as the training vectors defined above: Pd, Tl, Ai, Q, and TS. Note that the validation and testing Q and TS values can be different from each other and from those used by the training vectors. The dimensions above have the following definitions: • N_l is the number of network layers (net.numLayers). • N_i is the number of network inputs (net.numInputs). • R^j is the size of the jth input (net.inputs{j}.size). • S^i is the size of the ith layer (net.layers{i}.size) • LD is the number of layer delays (net.numLayerDelays). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (length(net.inputWeights{i,j}.delays)). Your training function must also provide information about itself using this calling format, `info = yourtf(code)` where the correct information is returned for each of the following string codes: • 'version' - Returns the Neural Network Toolbox version (3.0). • 'pdefaults' - Returns a structure of default training parameters. 402.

U.S. Patent No. RE49,461	nnet
	When you set the network training function (`net.trainFcn`) to be your function, the network's training parameters (`net.trainParam`) automatically is set to your default structure. Those values can be altered (or not) before training. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size. For performance reasons, `train` turns off the normal type checking for network properties before calling your training function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or adapt the network. If you are interested in creating your own training function, you can examine the implementations of toolbox functions such as `trainc` and `trainr`. The help for each of these utility functions lists the input and output arguments they take. **Utility Functions.** If you examine training functions such as `trainc`, `traingd`, and `trainlm`, note that they use a set of utility functions found in the `nnet/nnutils` directory. These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you might have to update your functions for future versions of the toolbox. Use `help` on each function to view the function's input and output arguments. These two functions are useful for creating a new training record and truncating it once the final number of epochs is known: • `newtr` - New training record with any number of optional fields. • `cliptr` - Clip training record to the final number of epochs. These three functions calculate network signals going forward, errors, and derivatives of performance coming back: • `calca` - Calculate network outputs and other signals. • `calcerr` - Calculate matrix or cell array errors. • `calcgrad` - Calculate bias and weight performance gradients. 403.

U.S. Patent No. RE49,461	nnet

	These two functions get and set a network's weight and bias values with single vectors. Being able to treat all these adjustable parameters as a single vector is often useful for implementing optimization algorithms:
	• `getx` - Get all network weight and bias values as a single vector.
	• `setx` - Set all network weight and bias values with a single vector.
	These next three functions are also useful for implementing optimization functions. One calculates all network signals going forward, including errors and performance. One backpropagates to find the derivatives of performance as a single vector. The third function backpropagates to find the Jacobian of performance. This latter function is used by advanced optimization techniques like Levenberg-Marquardt:
	• `calcperf` - Calculate network outputs, signals, and performance.
	• `calcgx` - Calculate weight and bias performance gradient as a single vector.
	• `calcjx` - Calculate weight and bias performance Jacobian as a single matrix.
	Adapt Functions
	The other kind of the general learning function is a network adapt function. Adapt functions simulate a network, while updating the network for each time step of the input before continuing the simulation to the next input.
	Once defined, you can assign your adapt function to a network.
	`net.adaptFcn = 'youraf';`
	Your network initialization function is used whenever you adapt your network.
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid adapt function, it must take and return a network,
	`[net,Ac,El] = youraf(net,Pd,Tl,Ai,Q,TS)`
	where:
	• `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
	- Each `Pd{i,j,ts}` is the $R^j \times (D_i{}^{ij}Q)$ delayed input matrix to the weight going to the ith layer from the jth input at time step `ts`. Note that (`Pd{i,j,ts}` is an empty matrix `[]` if the ith layer doesn't have a weight from the jth input.)
	404.

U.S. Patent No. RE49,461	nnet
	• `Tl` is an $N_l \times TS$ cell array of layer targets. - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. Note that (`Tl{i,ts}` is an empty matrix if the ith layer doesn't have a target.) • `Ai` is an $N_l \times LD$ cell array of initial layer delay states. - Each `Ai{l,k}` is the $S^i \times Q$ delayed ith layer output for time step `ts = k`-LD, where ts goes from 0 to LD-1. • `Q` is the number of concurrent vectors. • `TS` is the number of time steps. The dimensions above have the following definitions: • N_l is the number of network layers (`net.numLayers`). • N_i is the number of network inputs (`net.numInputs`). • R^j is the size of the jth input (`net.inputs{j}.size`). • S^i is the size of the ith layer (`net.layers{i}.size`) • LD is the number of layer delays (`net.numLayerDelays`). • D_i^{ij} is the number of delay lines associated with the weight going to the ith layer from the jth input (`length(net.inputWeights{i,j}.delays)`). Your adapt function must also provide information about itself using this calling format, `info = youraf(code)` where the correct information is returned for each of the following string codes: • `'version'` - Returns the Neural Network Toolbox version (3.0). • `'pdefaults'` - Returns a structure of default adapt parameters. When you set the network adapt function (`net.adaptFcn`) to be your function, the network's adapt parameters (`net.adaptParam`) automatically is set to your default structure. Those values can then be altered (or not) before adapting. Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `adapt` turns off the normal type checking for network properties before calling your adapt function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or train the network. 405.

U.S. Patent No. RE49,461	nnet

	If you are interested in creating your own training function, you can examine the implementation of a toolbox function such as `trains`.
	
	Utility Functions. If you examine the toolbox's only adapt function `trains`, note that it uses a set of utility functions found in the `nnet/nnutils` directory. The help for each of these utility functions lists the input and output arguments they take.
	
	These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you will have to update your functions for future versions of the toolbox.
	
	These two functions are useful for simulating a network, and calculating its derivatives of performance:
	
	• `calca1` - New training record with any number of optional fields.
	• `calce1` - Clip training record to the final number of epochs.
	• `calcgrad` - Calculate bias and weight performance gradients.
	
	Performance Functions
	Performance functions allow a network's behavior to be graded. This is useful for many algorithms, such as backpropagation, which operate by adjusting network weights and biases to improve performance.
	
	Once defined you can assign your training function to a network.
	
	`net.performFcn = 'yourpf';`
	
	Your network initialization function will then be used whenever you train your adapt your network.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid performance function your function must be called as follows,
	
	`perf = yourpf(E,X,PP)`
	
	where:
	
	• `E` is either an S x Q matrix or an $N_l \times TS$ cell array of layer errors.
	
	406.

U.S. Patent No. RE49,461	nnet

	- Each `E{i,ts}` is the $S^i \times Q$ target matrix for the ith layer. (`Tl(i,ts)` is an empty matrix if the ith layer doesn't have a target.)
	• `X` is an M x 1 vector of all the network's weights and biases.
	• `PP` is a structure of network performance parameters.
	If E is a cell array you can convert it to a matrix as follows.
	`E = cell2mat(E);`
	Alternatively, your function must also be able to be called as follows,
	`perf = yourpf(E,net)`
	where you can get `X` and `PP` (if needed) as follows.
	`X = getx(net);`
	`PP = net.performParam;`
	Your performance function must also provide information about itself using this calling format,
	`info = yourpf(code)`
	where the correct information is returned for each of the following string codes:
	• `'version'` - Returns the Neural Network Toolbox version (3.0).
	• `'deriv'` - Returns the name of the associated derivative function.
	• `'pdefaults'` - Returns a structure of default performance parameters.
	When you set the network performance function (`net.performFcn`) to be your function, the network's adapt parameters (`net.performParam`) will automatically get set to your default structure. Those values can then be altered or not before training or adaption.
	To see how an example custom performance function works type in these lines of code.
	`help mypf`
	`e = rand(4,5);`
	`x = rand(12,1);`
	`pp = mypf('pdefaults')`
	`perf = mypf(e,x,pp)`
	
	407.

U.S. Patent No. RE49,461	nnet

	Use this command to see how mypf was implemented.
	`type mypf`
	You can use mypf as a template to create your own weight and bias initialization function.
	Performance Derivative Functions. If you want to use backpropagation with your performance function, you need to create a custom derivative function for it. It needs to calculate the derivative of the network's errors and combined weight and bias vector, with respect to performance,
	`dPerf_dE = dmsereg('e',E,X,perf,PP)`
	`dPerf_dX = dmsereg('x',E,X,perf,PP)`
	where:
	• E is an $N_l \times TS$ cell array of layer errors.
	- Each E{i,ts} is the $S^i \times Q$ target matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• X is an $M \times 1$ vector of all the network's weights and biases.
	• PP is a structure of network performance parameters.
	• dPerf_dE is the $N_l \times TS$ cell array of derivatives dPerf/dE.
	- Each E{i,ts} is the $S^i \times Q$ derivative matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the ith layer doesn't have a target.)
	• dPerf_dX is the $M \times 1$ derivative dPerf/dX.
	To see how the example custom performance derivative function mydpf works, type
	`help mydpf`
	`e = {e};`
	`dperf_de = mydpf('e',e,x,perf,pp)`
	`dperf_dx = mydpf('x',e,x,perf,pp)`
	Use this command to see how mydpf was implemented.
	`type mydpf`
	You can use mydpf as a template to create your own performance derivative functions.
	408.

U.S. Patent No. RE49,461	nnet

	Weight and Bias Learning Functions
	The most specific kind of learning function is a weight and bias learning function. These functions are used to update individual weights and biases during learning. with some training and adapt functions.
	
	Once defined. you can assign your learning function to any weight and bias in a network. For example, the following lines of code assign the weight and bias learning function `yourwblf` to the second layer's bias, and the weight coming from the first input to the second layer.
	
	`net.biases{2}.learnFcn = 'yourwblf';`
	`net.inputWeights{2,1}.learnFcn = 'yourwblf';`
	
	Weight and bias learning functions are only called to update weights and biases if the network training function (`net.trainFcn`) is set to `trainb`, `trainc`, or `trainr`, or if the network adapt function (`net.adaptFcn`) is set to `trains`. If this is the case, then your function is used to update the weight and biases it is assigned to whenever you train or adapt your network with `train` or `adapt`.
	
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	
	To be a valid weight and bias learning function, it must be callable as follows,
	
	`[dW,LS] = yourwblf(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)`
	
	where:
	
	• `W` is an $S \times R$ weight matrix.
	• `P` is an $R \times Q$ matrix of Q input (column) vectors.
	• `Z` is an $S \times Q$ matrix of Q weighted input (column) vectors.
	• `N` is an $S \times Q$ matrix of Q net input (column) vectors.
	• `A` is an $S \times Q$ matrix of Q layer output (column) vectors.
	• `T` is an $S \times Q$ matrix of Q target (column) vectors.
	• `E` is an $S \times Q$ matrix of Q error (column) vectors.
	• `gW` is an $S \times R$ gradient of `W` with respect to performance.
	• `gA` is an $S \times Q$ gradient of `A` with respect to performance.
	• `D` is an $S \times S$ matrix of neuron distances.
	• `LP` is a a structure of learning parameters.
	
	409.

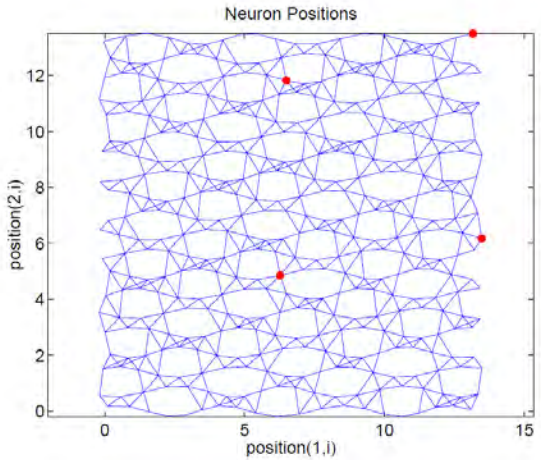
U.S. Patent No. RE49,461	nnet

	• LS is a structure of the learning state that is updated for each call. (Use a null matrix [ ] the first time.)
	• dW is the resulting $S \times R$ weight change matrix.
	Your function is called as follows to update bias vector
	`[db,LS] = yourwblf(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)`
	where:
	• S is the number of neurons in the layer.
	• b is a new $S \times 1$ bias vector.
	Your learning function must also provide information about itself using this calling format,
	`info = yourwblf(code)`
	where the correct information is returned for each of the following string codes:
	• 'version' - Returns the Neural Network Toolbox version (3.0).
	• 'deriv' - Returns the name of the associated derivative function.
	• 'pdefaults' - Returns a structure of default performance parameters.
	To see how an example custom weight and bias initialization function works, type
	`help mywblf`
	Use this command to see how mywbif was implemented.
	`type mywblf`
	You can use mywblf as a template to create your own weight and bias learning function.
	Self-Organizing Map Functions
	There are two kinds of functions that control how neurons in self-organizing maps respond. They are topology and distance functions.
	Topology Functions
	Topology functions calculate the positions of a layer's neurons given its dimensions.
	410.

U.S. Patent No. RE49,461	nnet

	Once defined, you can assign your topology function to any layer of a network. For example, the following line of code assigns the topology function `yourtopf` to the second layer of a network.
	`net.layers{2}.topologyFcn = 'yourtopf';`
	Your topology function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	To be a valid topology function your function must calculate positions `pos` from dimensions `dim` as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• `dimi` is the number of neurons along the ith dimension of the layer.
	• `pos` is an $N \times S$ matrix of S position vectors, where S is the total number of neurons that is defined by the product `dim1*dim1*...*dimN`.
	The toolbox contains an example custom topology function called `mytopf`. Enter the following lines of code to see how it is used.
	`help mytopf`
	`pos = mytopf(20,20);`
	`plotsom(pos)`
	If you type that code, you get the following plot.
	
	411.

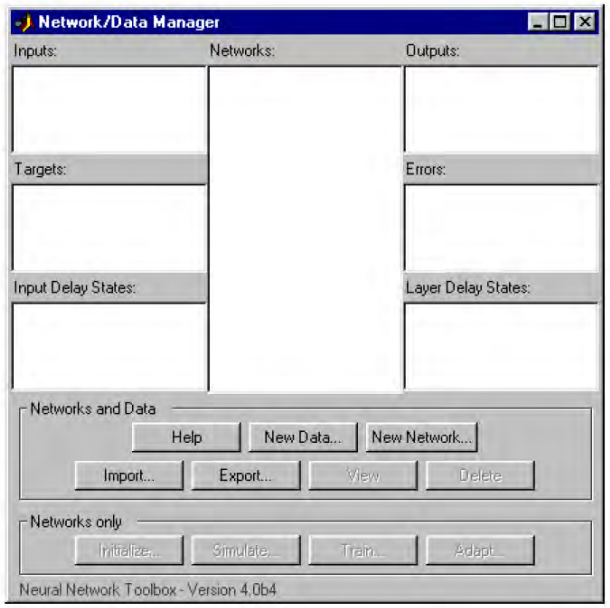
U.S. Patent No. RE49,461	nnet

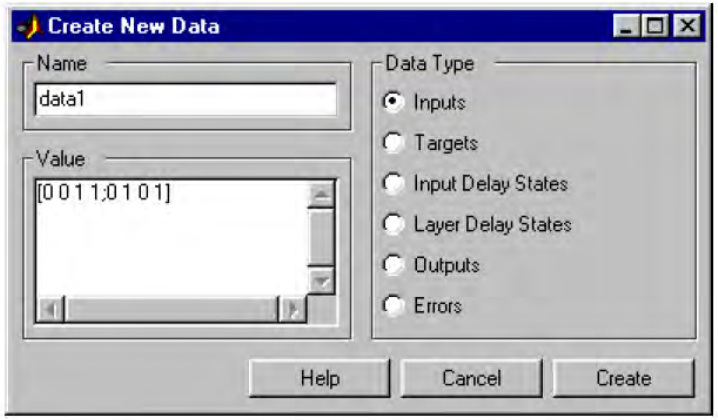
	
	Enter the following command to see how `mytf` is implemented.
	`type mytopf`
	You can use `mytopf` as a template to create your own topology function.
	Distance Functions
	Distance functions calculate the distances of a layer's neurons given their position.
	Once defined, you can assign your distance function to any layer of a network. For example, the following line of code assigns the topology function `yourdistf` to the second layer of a network.
	`net.layers{2}.distanceFcn = 'yourdistf';`
	Your distance function is used whenever your network is trained or adapts.
	`[net,tr] = train(NET,P,T,Pi,Ai)`
	`[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)`
	412.

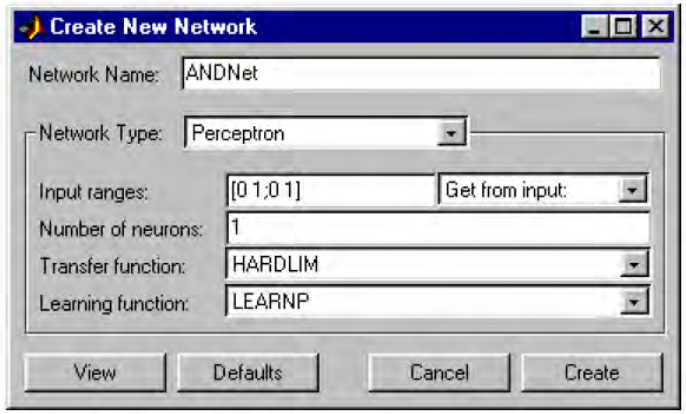
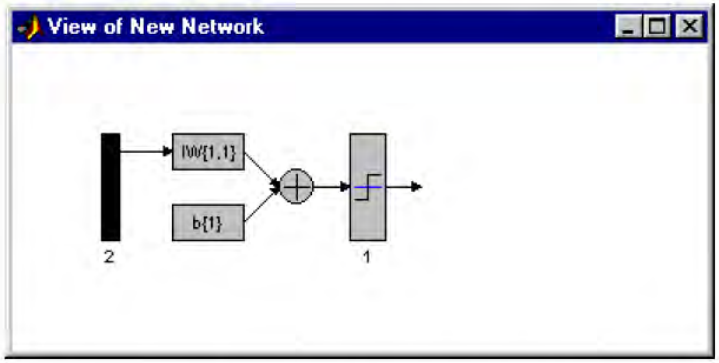
U.S. Patent No. RE49,461	nnet

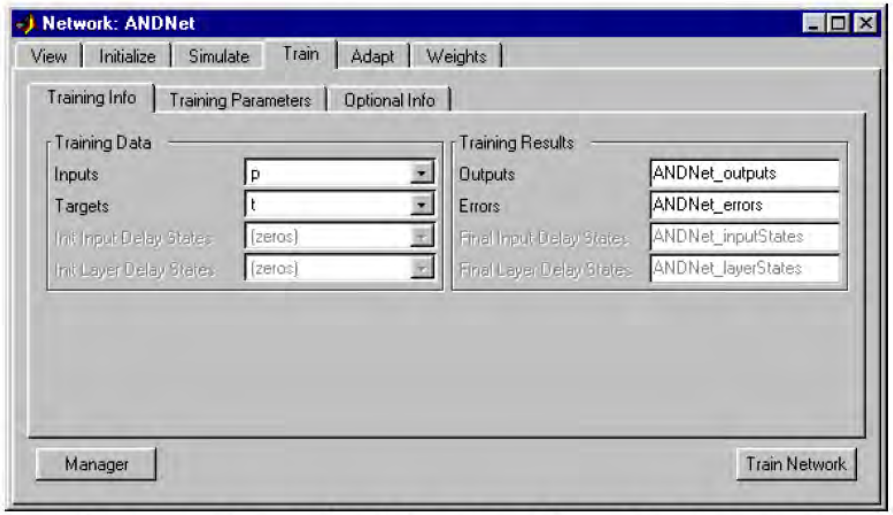
	To be a valid distance function, it must calculate distances d from position pos as follows,
	`pos = yourtopf(dim1,dim2,...,dimN)`
	where:
	• pos is an $N \times S$ matrix of S neuron position vectors.
	• d is an $S \times S$ matrix of neuron distances.
	The toolbox contains an example custom distance function called mydistf. Enter the following lines of code to see how it is used.
	`help mydistf`
	`pos = gridtop(4,5);`
	`d = mydistf(pos)`
	Enter the following command to see how mytf is implemented.
	`type mydistf`
	You can use mydistf as a template to create your own distance function.
	
	413.
	
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the

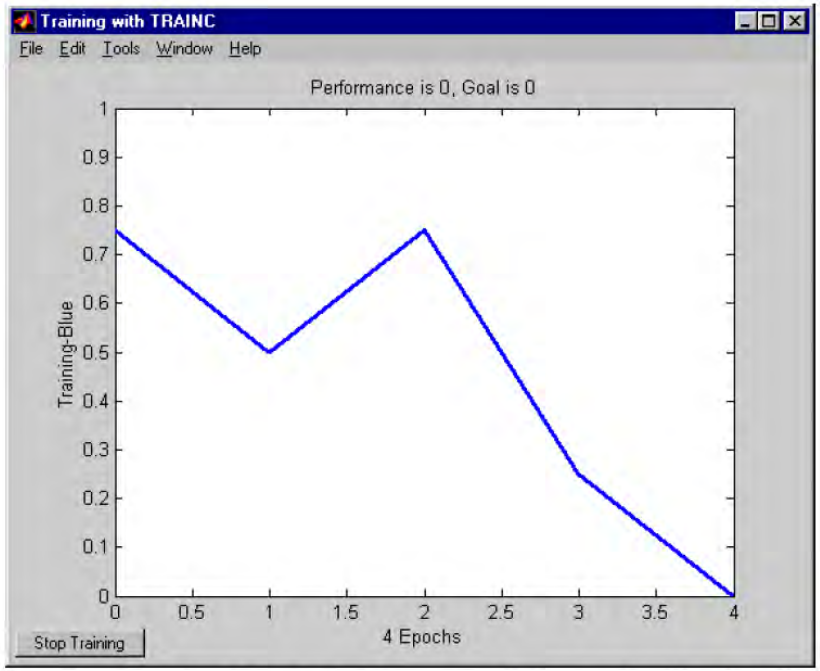
U.S. Patent No. RE49,461	nnet
	teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
26. The method of claim 21, wherein executing the user interaction stream comprises: specifying an output to be saved to disk and/or displayed on a screen.	nnet discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises specifying an output to be saved to disk and/or displayed on a screen. *See* claim 21; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses specifying an output to be saved to disk and or displayed on a screen (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

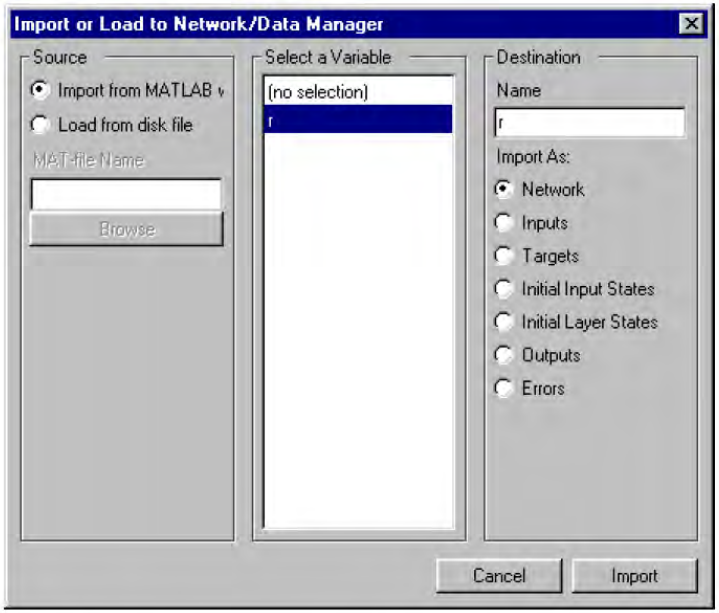
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

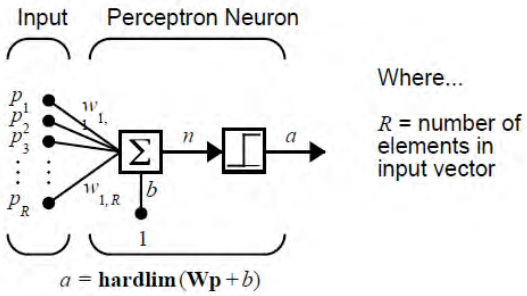
U.S. Patent No. RE49,461	nnet

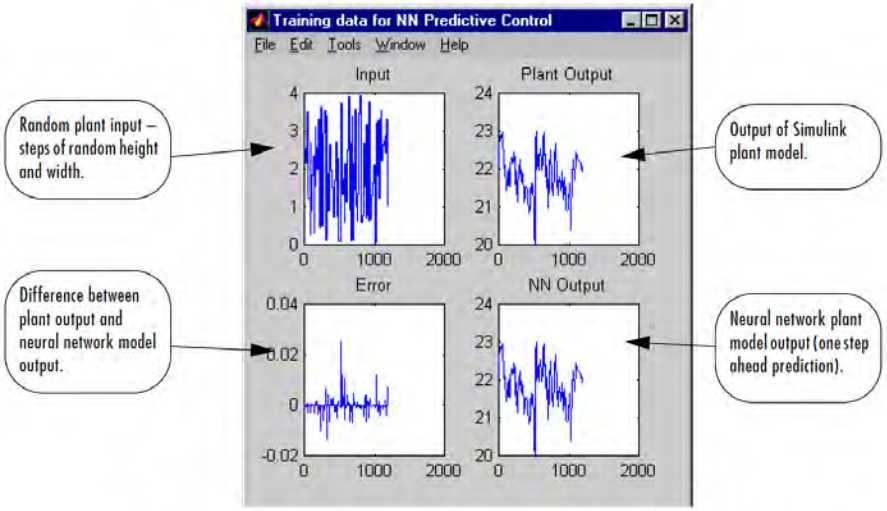
	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

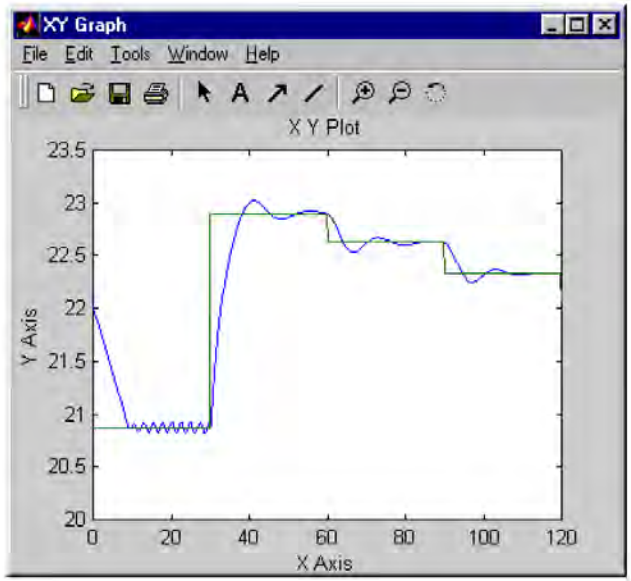
U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

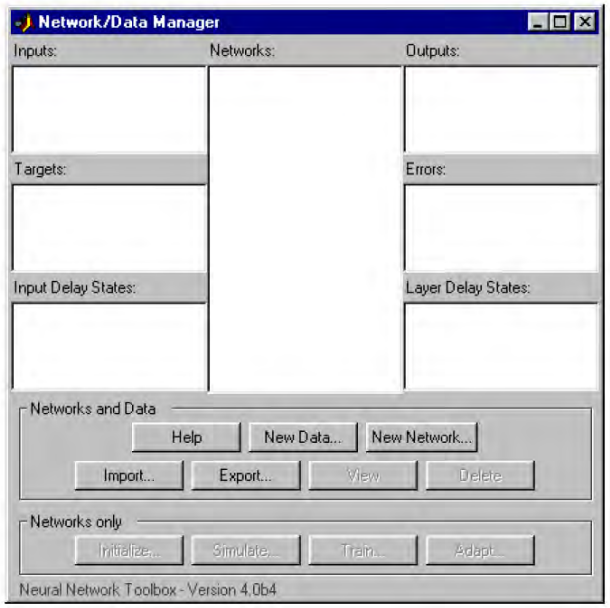
U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

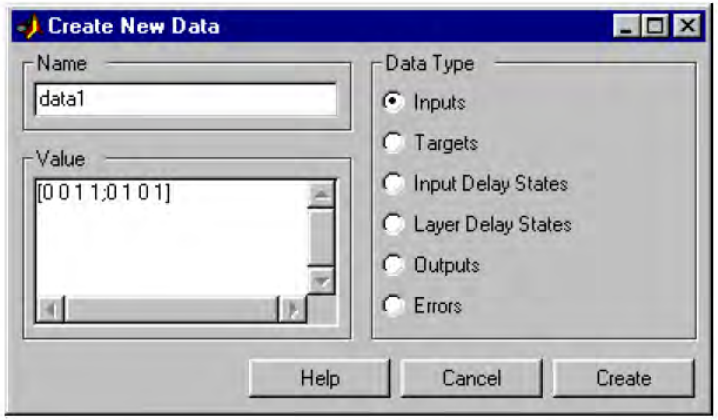
U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  $a = \mathbf{hardlim}(\mathbf{Wp} + b)$ 101.

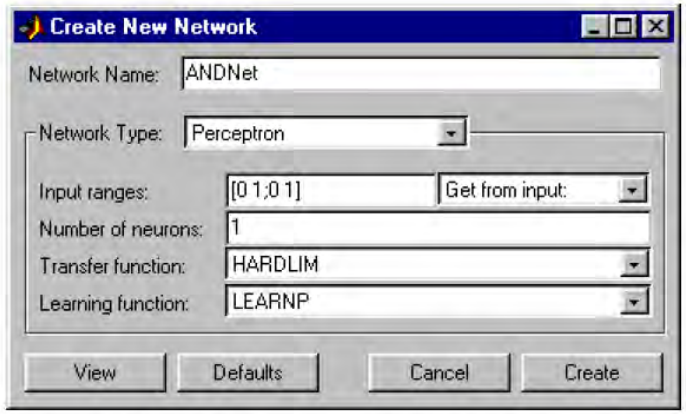
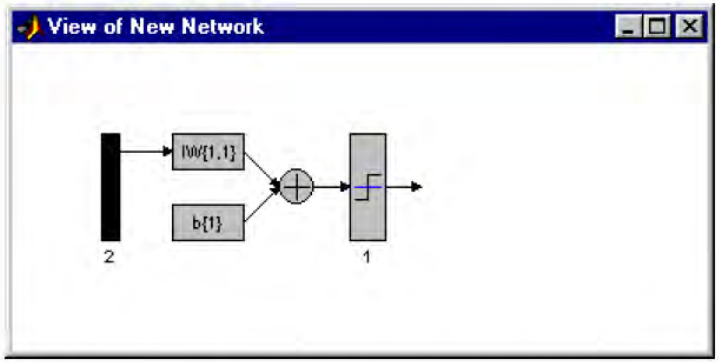
U.S. Patent No. RE49,461	nnet
218.	according to the selected training algorithm (`trainlm` in this case). This is a straightforward application of batch training, as described in Chapter 5, "Backpropagation." After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.) You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this demonstration, begin the simulation, as shown in the following steps.  **7** Select **OK** in the **Plant Identification** window. This loads the trained neural network plant model into the NN Predictive Controller block. **8** Select **OK** in the **Neural Network Predictive Control** window. This loads the controller parameters into the NN Predictive Controller block. **9** Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the

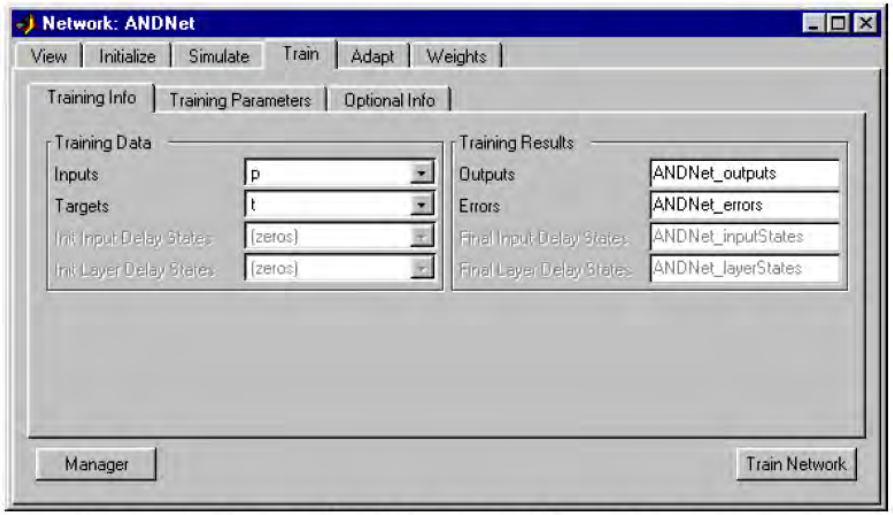
U.S. Patent No. RE49,461	nnet
	An nnet user can change display properties of an output of the computations by using standard interface functions like zooming in and out, rotating a plot, and adding labels. plant output and the reference signal are displayed, as in the following figure.  219. To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any

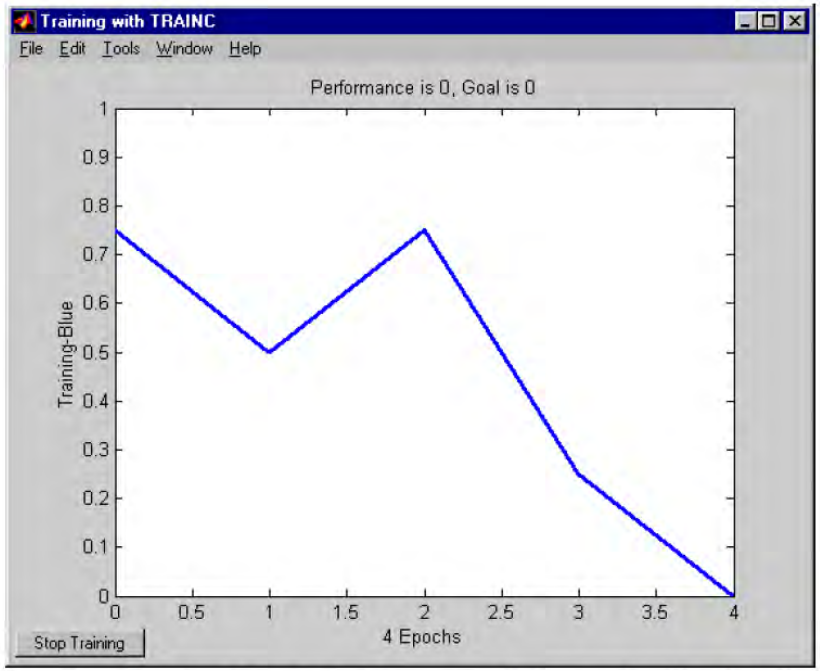
U.S. Patent No. RE49,461	nnet
	supplements thereto and the relevant section of charts for other prior art for the ’461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
27. The method of claim 21, wherein executing the user interaction stream comprises: parsing elements to be used in the computations representing the artificial neural network.	nnet discloses, expressly and/or inherently, the method of claim 21, wherein executing the user interaction stream comprises parsing elements to be used in the computations representing the artificial neural network. *See* claim 21; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses parsing elements to be used in the computations representing the artificial neural network (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
  2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
  ans =
  -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

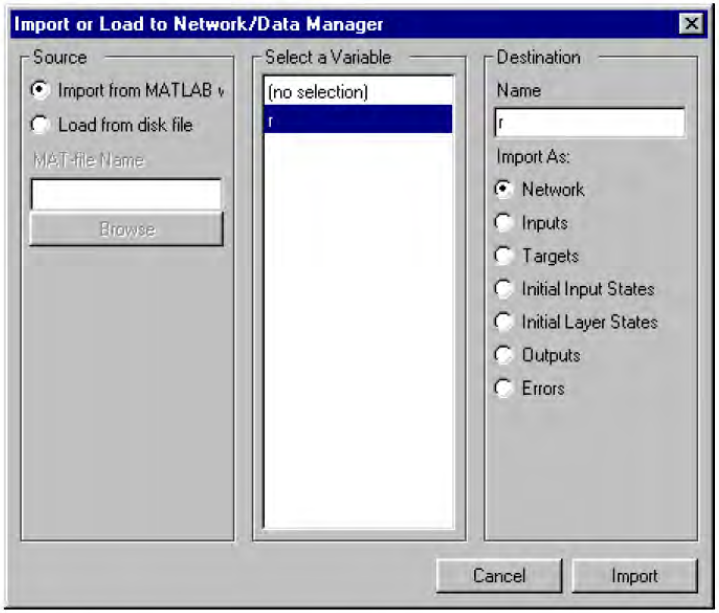
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

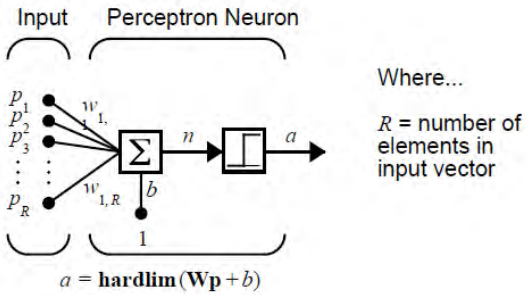
Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

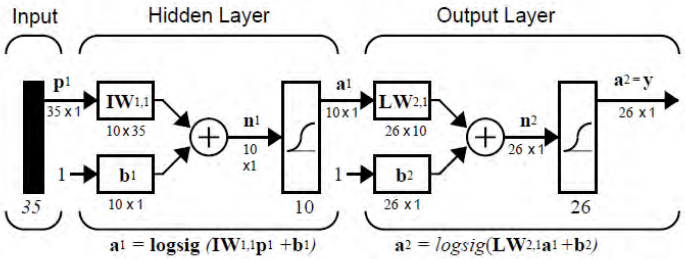
Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

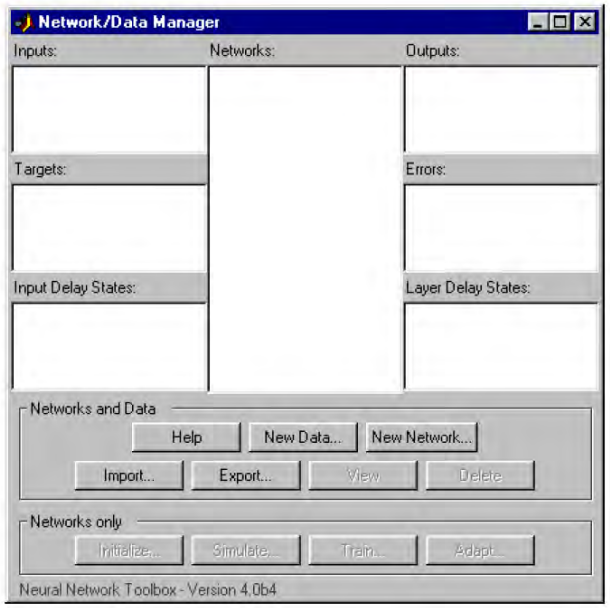
U.S. Patent No. RE49,461	nnet

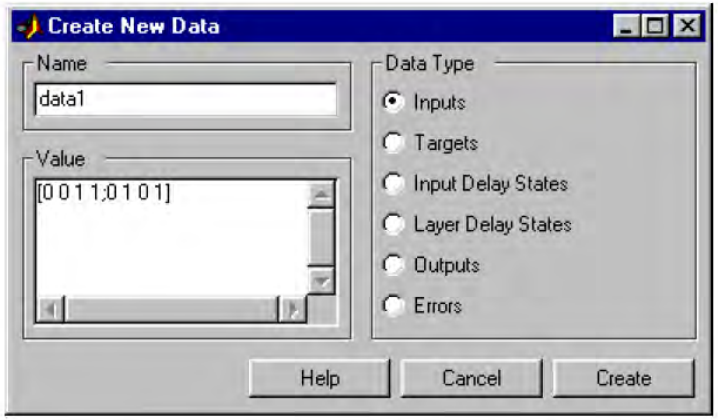
	network. The log-sigmoid transfer function was picked because its output range (0 to 1) is perfect for learning to output boolean values.
	 
	
	The hidden (first) layer has 10 neurons. This number was picked by guesswork and experience. If the network has trouble learning, then neurons can be added to this layer.
	The network is trained to output a 1 in the correct position of the output vector and to fill the rest of the output vector with 0's. However, noisy input vectors may result in the network not creating perfect 1's and 0's. After the network is trained the output is passed through the competitive transfer function `compet`. This makes sure that the output corresponding to the letter most like the noisy input vector takes on a value of 1, and all others have a value of 0. The result of this post-processing is the output that is actually used.
	Initialization
	The two-layer network is created with `newff`.
	`S1 = 10;`
	`[R,Q] = size(alphabet);`
	`[S2,Q] = size(targets);`
	`P = alphabet;`
	`net = newff(minmax(P),[S1 S2],{'logsig' 'logsig'},'traingdx');`
	Training
	To create a network that can handle noisy input vectors it is best to train the network on both ideal and noisy vectors. To do this, the network is first trained on ideal vectors until it has a low sum-squared error.
	
	370.

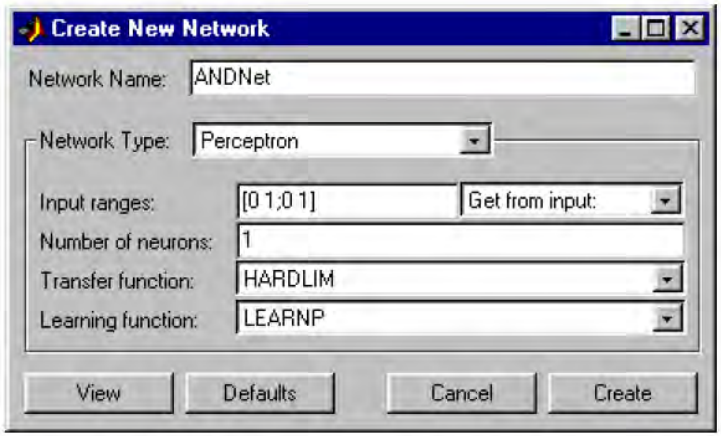
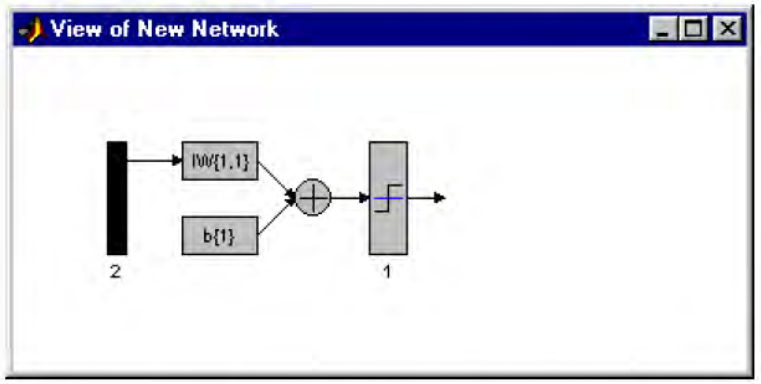
U.S. Patent No. RE49,461	nnet

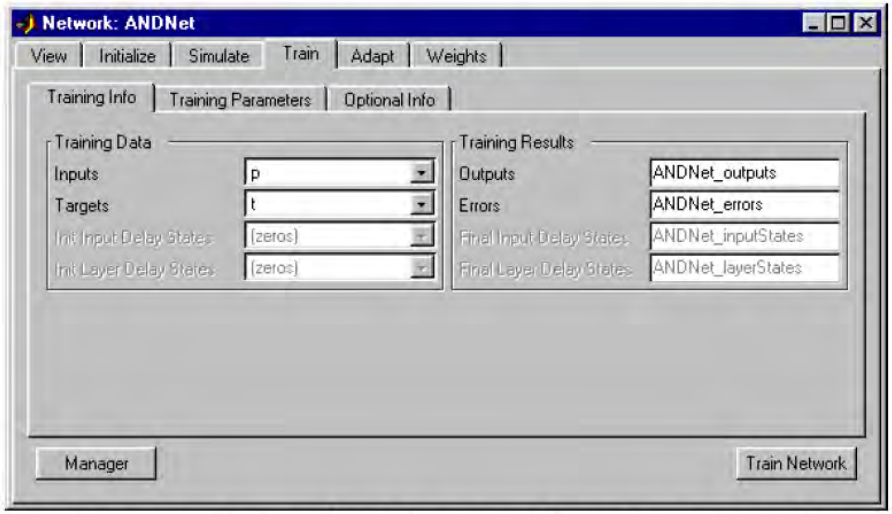
	Then, the network is trained on 10 sets of ideal and noisy vectors. The network is trained on two copies of the noise-free alphabet at the same time as it is trained on noisy vectors. The two copies of the noise-free alphabet are used to maintain the network's ability to classify ideal input vectors.
	
	Unfortunately, after the training described above the network may have learned to classify some difficult noisy vectors at the expense of properly classifying a noise-free vector. Therefore, the network is again trained on just ideal vectors. This ensures that the network responds perfectly when presented with an ideal letter.
	
	All training is done using backpropagation with both adaptive learning rate and momentum with the function `trainbpx`.
	
	Training Without Noise
	The network is initially trained without noise for a maximum of 5000 epochs or until the network sum-squared error falls beneath 0.1.
	
	`P = alphabet;`
	`T = targets;`
	`net.performFcn = 'sse';`
	`net.trainParam.goal = 0.1;`
	`net.trainParam.show = 20;`
	`net.trainParam.epochs = 5000;`
	`net.trainParam.mc = 0.95;`
	`[net,tr] = train(net,P,T);`
	
	Training with Noise
	To obtain a network not sensitive to noise, we trained with two ideal copies and two noisy copies of the vectors in `alphabet`. The target vectors consist of four copies of the vectors in `target`. The noisy vectors have noise of mean 0.1 and 0.2 added to them. This forces the neuron to learn how to properly identify noisy letters, while requiring that it can still respond well to ideal vectors.
	
	To train with noise, the maximum number of epochs is reduced to 300 and the error goal is increased to 0.6, reflecting that higher error is expected because more vectors (including some with noise), are being presented.
	
	`netn = net;`
	`netn.trainParam.goal = 0.6;`
	`netn.trainParam.epochs = 300;`
	
	371.

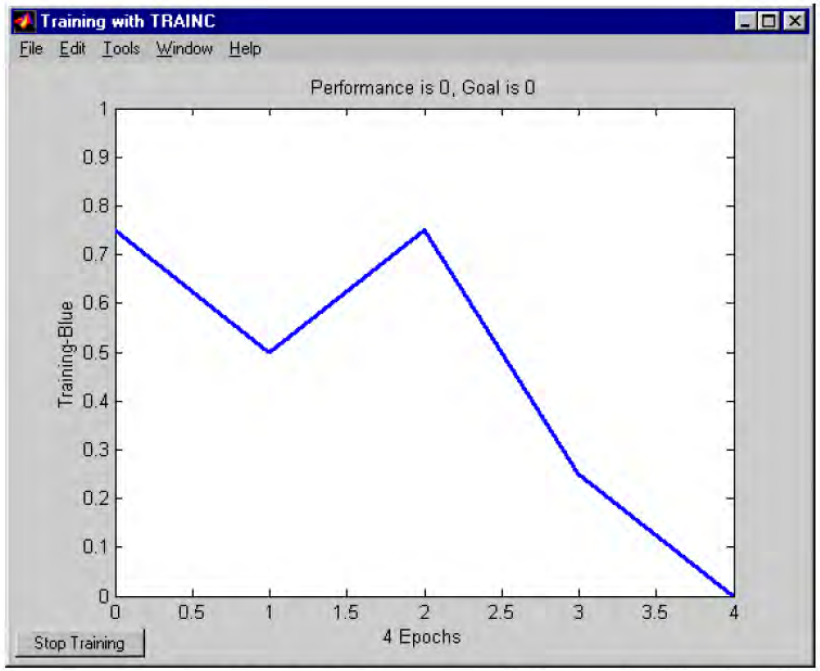
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
28. The method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises: converting the elements into GPU programs.	nnet discloses, expressly and/or inherently, the method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises converting the elements into GPU programs. *See* claim 27; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses converting elements into GPU programs (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

U.S. Patent No. RE49,461	nnet
95.	**Train the Perceptron** To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

U.S. Patent No. RE49,461	nnet
	98.

```
   2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
   ans =
   -3.
```

Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

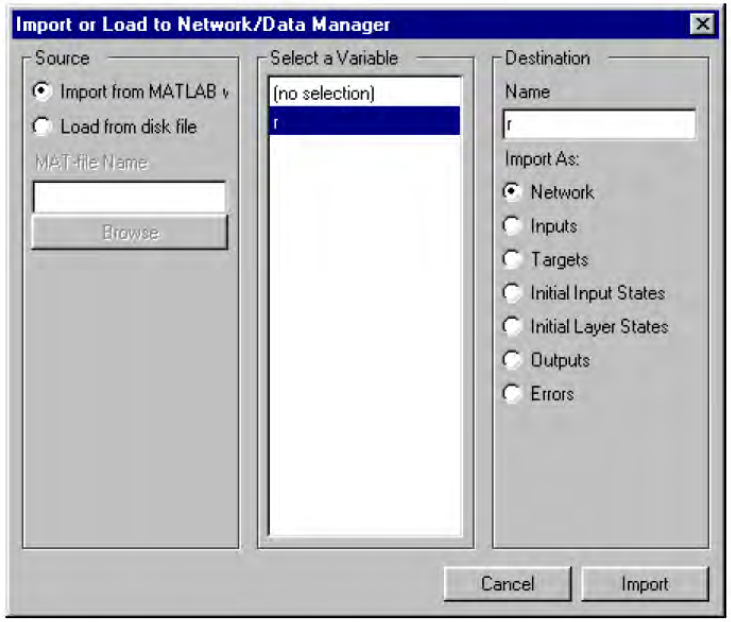
Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

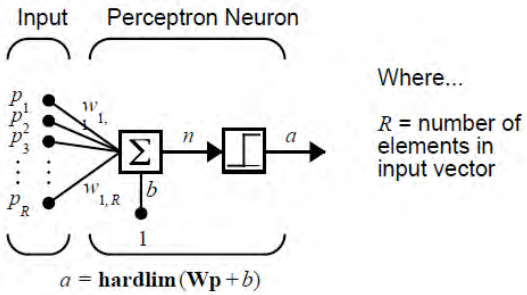
Create a new vector.

```
r= [0; 1; 2; 3]
r =
     0
     1
     2
     3
```

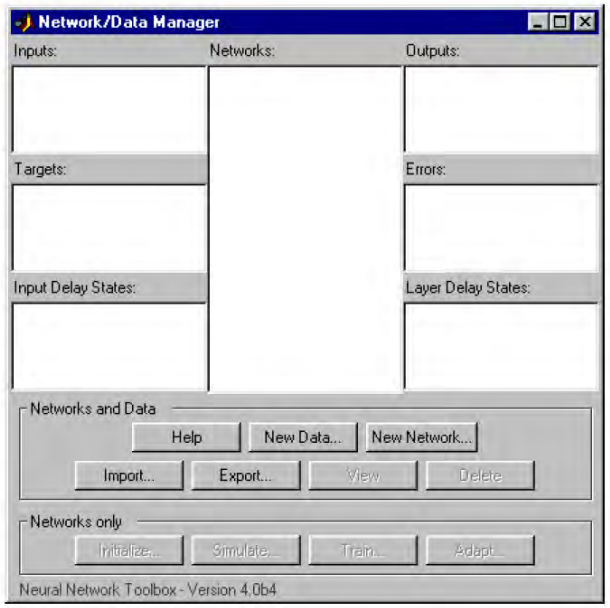
Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this

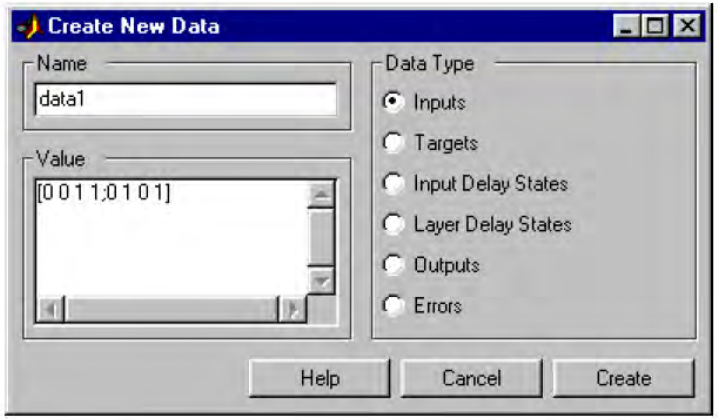
U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

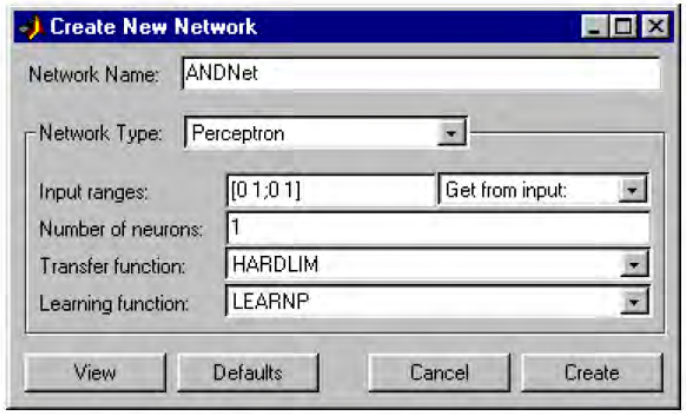
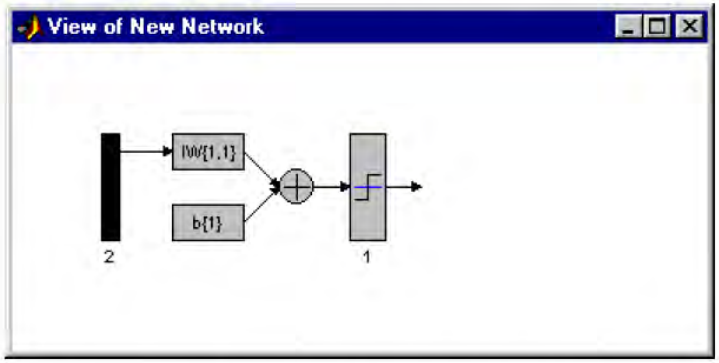
U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

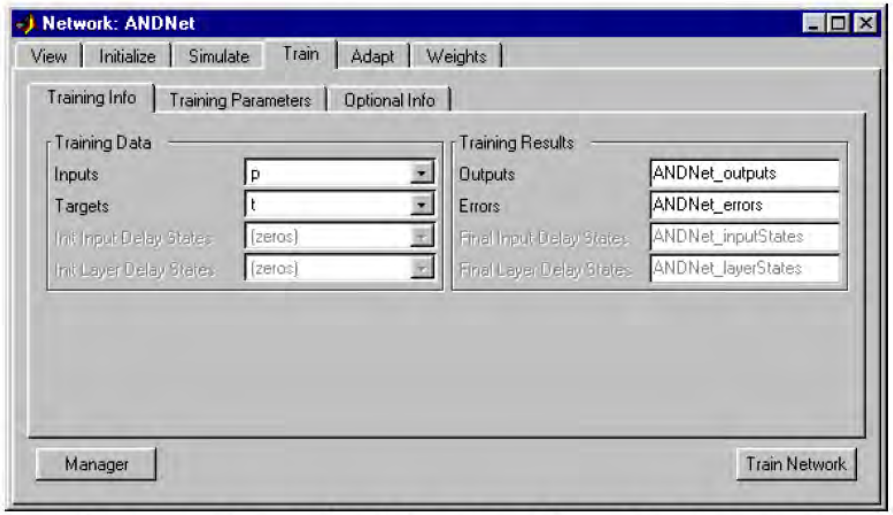
U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

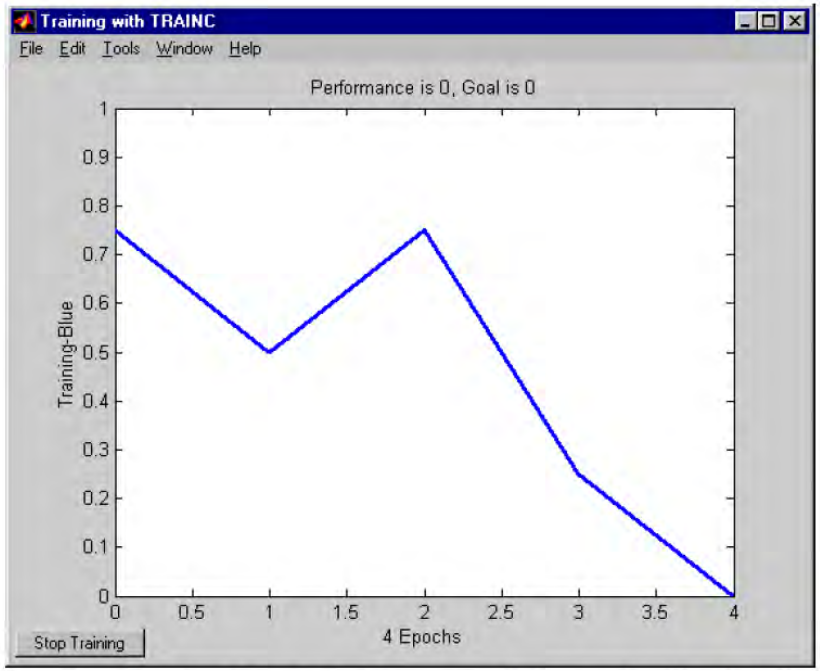
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
29. The method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs.	nnet discloses, expressly and/or inherently, the method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs. *See* claim 28; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses compiling the GPU programs (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

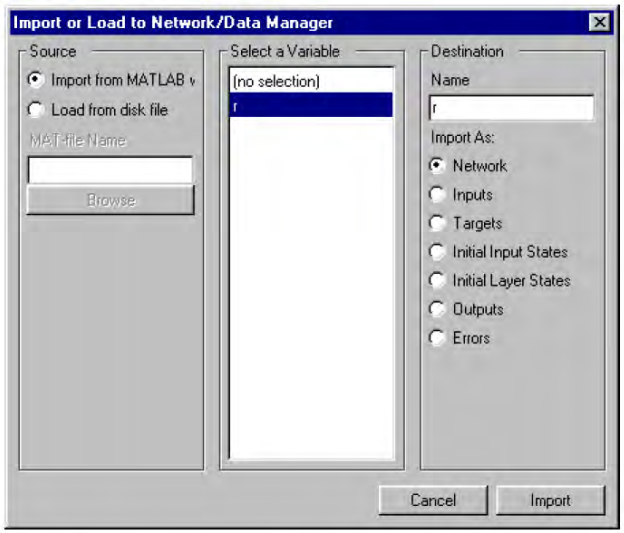
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

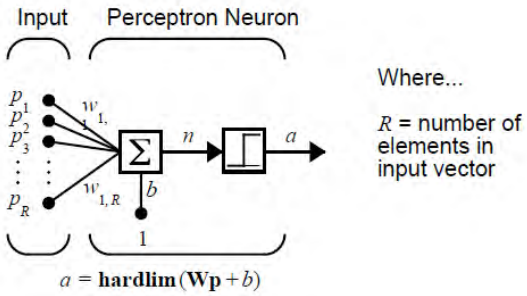
U.S. Patent No. RE49,461	nnet

	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

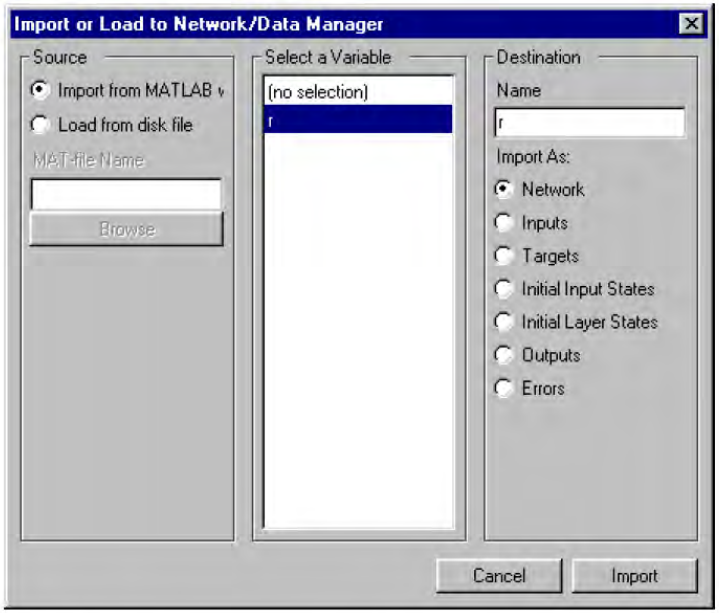
U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

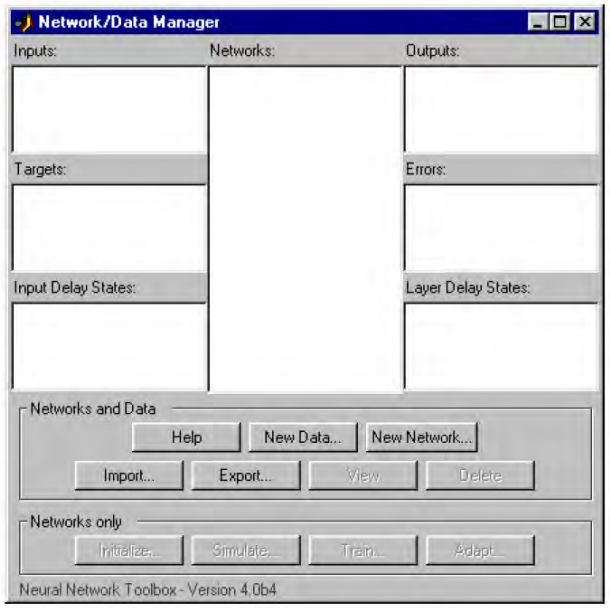
U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

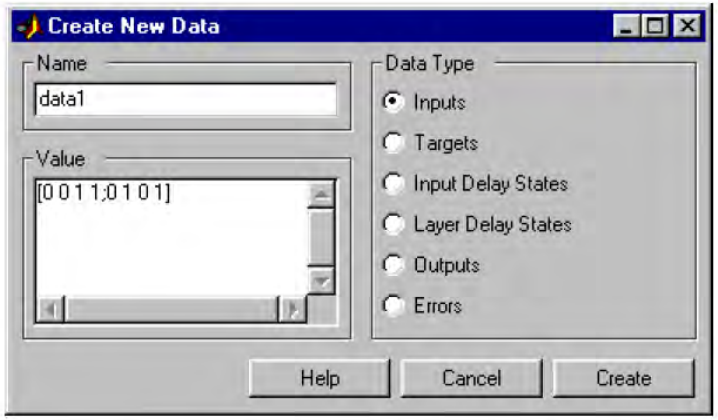
U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

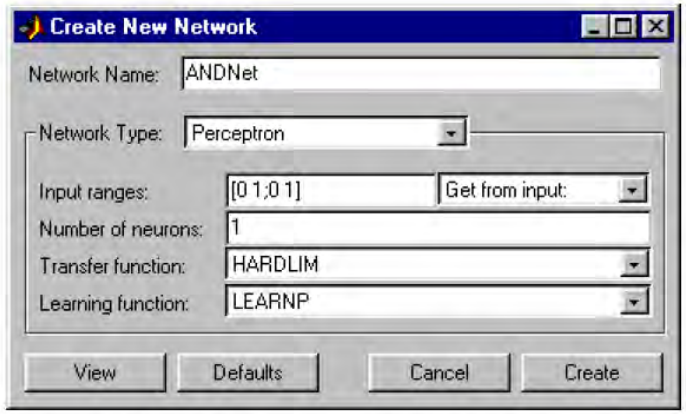
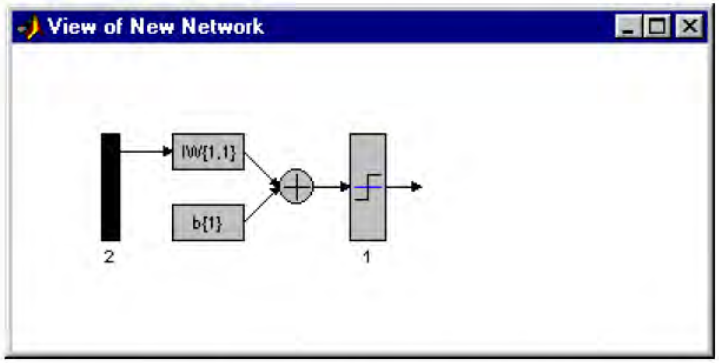
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
30. The method of claim 29, wherein executing the user interaction stream comprises: transferring the GPU programs to the second memory partition.	nnet discloses, expressly and/or inherently, the method of claim 29, wherein executing the user interaction stream comprises transferring the GPU programs to the second memory partition. *See* claim 29; Chapters 2-3, 6, 8, 10. *See e.g.*:

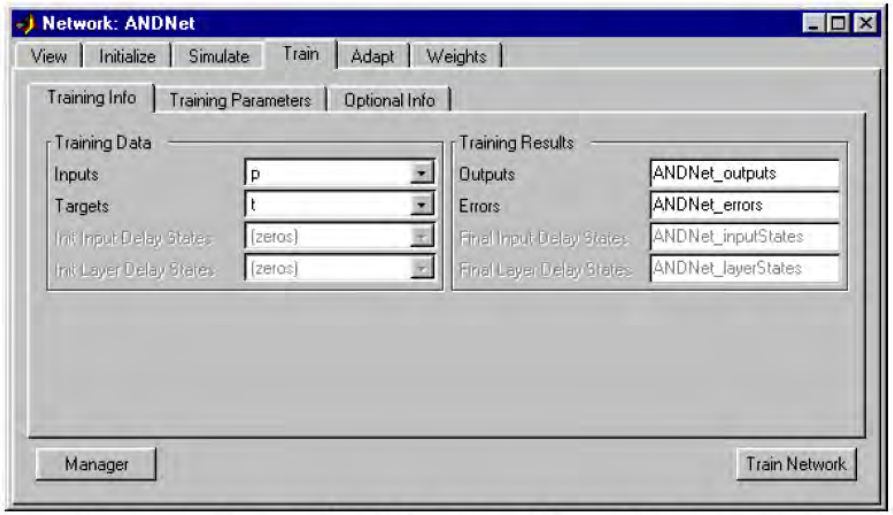
U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

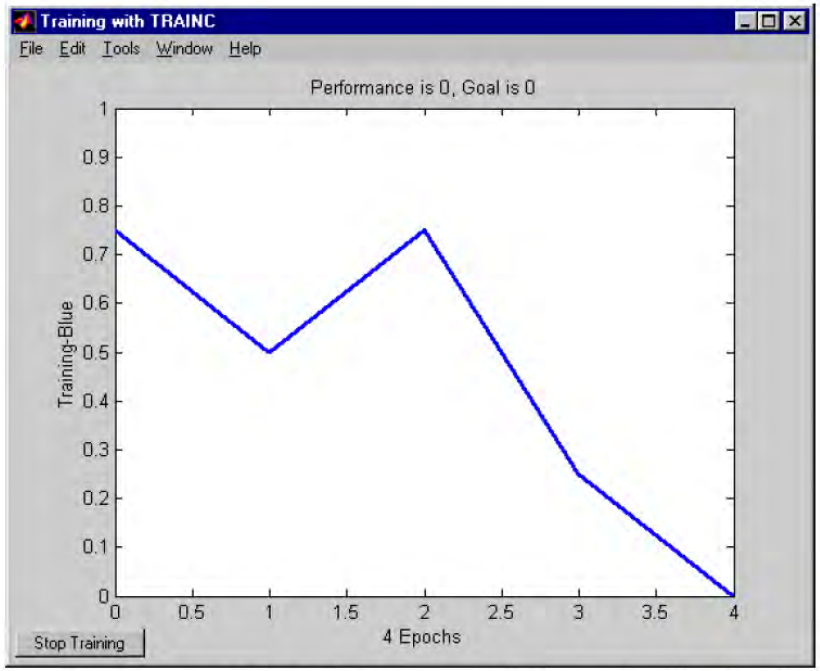
U.S. Patent No. RE49,461	nnet
	To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.
31. The method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk.	nnet discloses, expressly and/or inherently, the method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk. *See* claim 21; Chapters 2-3, 6, 8, 10. *See e.g.*: *As a non-limiting example, nnet discloses controlling transfer of outputs of the computations representing the artificial neural network to disk (e.g., nntool).*

U.S. Patent No. RE49,461	nnet
	 Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists. First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this: 92.

U.S. Patent No. RE49,461	nnet
	 Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input. Next we create a network target. Click on **New Data** again, and this time enter the variable name t, specify the value [0 0 0 1], and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input. **Create Network** Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter ANDNet under **Network Name**. Set the **Network Type** to Perceptron, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges [0 1;0 1].We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like: 93.

U.S. Patent No. RE49,461	nnet
	 Next you might look at the network by clicking on **View**. For example:  This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted. Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network. 94.

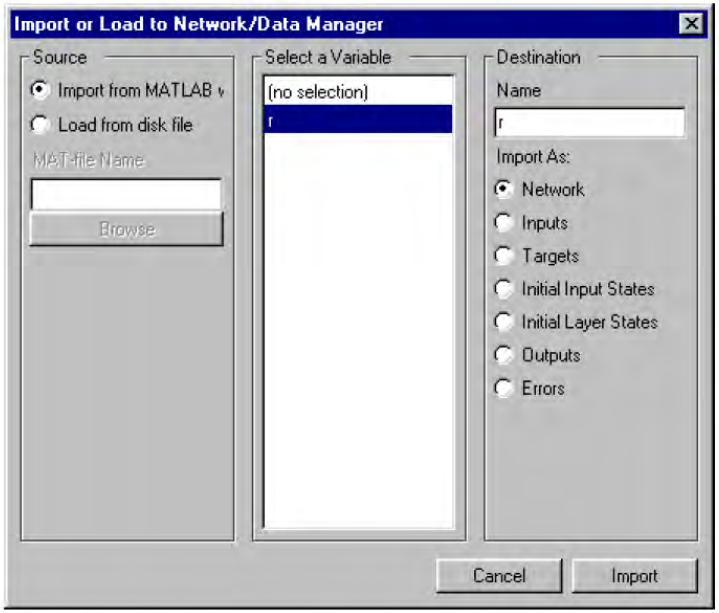
U.S. Patent No. RE49,461	nnet
	Train the Perceptron To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:  Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line. While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want. Now click **Train Network** to train the perceptron network. You will see the following training results. 95.

U.S. Patent No. RE49,461	nnet
	 Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.) You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate**). This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`. 96.

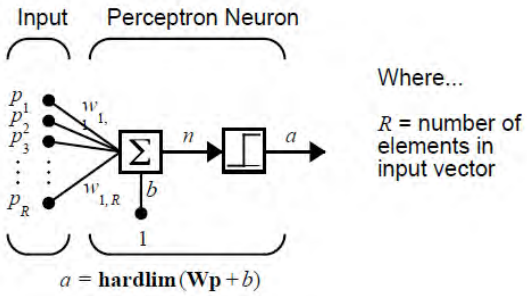
U.S. Patent No. RE49,461	nnet

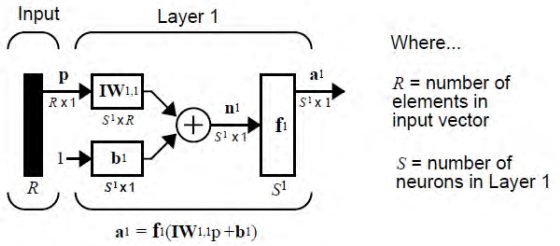
	Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value
	`[0 0 0 1]`
	Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.
	Export Perceptron Results to Workspace
	To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the `ANDNet` are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on `ANDNet_outputs` and `ANDNet_errors` to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type `who` to see all the defined variables. The result should be
	`who`
	`Your variables are:`
	`ANDNet_errors  ANDNet_outputs`
	You might type `ANDNet_outputs` and `ANDNet_errors` to obtain the following
	`ANDNet_outputs =`
	`0  0  0  1`
	and
	`ANDNet_errors =`
	`0  0  0  0.`
	You can export `p`, `t`, and `ANDNet` in a similar way. You might do this and check with `who` to make sure that they got to the command line.
	Now that `ANDNet` is exported you can view the network description and examine the network weight matrix. For instance, the command
	`ANDNet.iw{1,1}`
	gives
	`ans =`
	97.

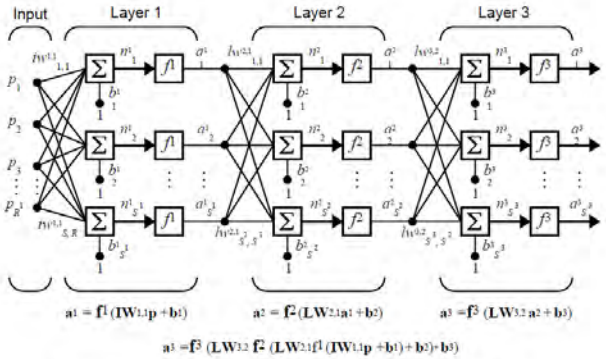
U.S. Patent No. RE49,461	nnet
	2 1 Similarly, ANDNet.b{1} yields ans = -3. **Clear Network/Data Window** You can clear the **Network/Data Manager** window by highlighting a variable such as p and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate. Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by nntool, gives a clean **Network/Data Manager** window. Recall however, that we exported p, t, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**. **Importing from the Command Line** To make thing simple, quit MATLAB. Start it again, and type nntool to begin a new session. Create a new vector. r= [0; 1; 2; 3] r = 0 1 2 3 Now click on **Import**, and set the destination **Name** to R (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this 98.

U.S. Patent No. RE49,461	nnet
	 Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable `R` is there as an input. **Save a Variable to a File and Load It Later** Bring up the **Network/Data Manager** and click on **New Network**. Set the name to `mynet`. Click on **Create.** The network name `mynet` should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select `mynet` in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file `mynetfile`. Now lets get rid of `mynet` in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight `mynet`, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** 99.

U.S. Patent No. RE49,461	nnet
	window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window. 100.

U.S. Patent No. RE49,461	nnet
	Summary Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem. The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem. Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution. Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors. A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace. **Figures and Equations** **Perceptron Neuron**  101.

U.S. Patent No. RE49,461	nnet
46.	**Inputs and Layers** We are about to discuss networks having multiple layers so we will need to extend our notation to talk about such networks. Specifically, we need to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. We also need to identify the source and destination for the weight matrices. We will call weight matrices connected to inputs, *input weights;* and we will call weight matrices coming from layer outputs, *layer weights.* Further, we will use superscripts to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, we have taken the one-layer multiple input network shown earlier and redrawn it in abbreviated form below.  As you can see, we have labeled the weight matrix connected to the input vector **p** as an Input Weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Also, elements of layer one, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer. In the next section, we will use Layer Weight (**LW**) matrices as well as Input Weight (**IW**) matrices. You might recall from the notation section of the Preface that conversion of the layer weight matrix from math to code for a particular network called *net* is: $\mathbf{IW}^{1,1} \rightarrow$ net.IW{1, 1} Thus, we could write the code to obtain the net input to the transfer function as:

U.S. Patent No. RE49,461	nnet
	n{1} = net.IW{1,1}*p + net.b{1}; **Multiple Layers of Neurons** A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in our figures, we append the number of the layer as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown below, and in the equations at the bottom of the figure.  $\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1)$ $\mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1 + \mathbf{b}^2)$ $\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2 + \mathbf{b}^3)$ $\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$ The network shown above has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the biases for each neuron. Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S^1 inputs, S^2 neurons, and an $S^2 \times S^1$ weight matrix $\mathbf{W}^2$. The input to layer 2 is $\mathbf{a}^1$; the output 47. To the extent Plaintiff contends that this claim element is not expressly or inherently disclosed, or is not obvious in view of nnet and the knowledge of a POSITA, it would have been obvious to combine the teachings of nnet with any of the prior art references in NVIDIA's Invalidity Contentions and any supplements thereto and the relevant section of charts for other prior art for the '461 patent in a manner that would result in the subject matter of this limitation given, at the very least, the person's understanding of the state of the art, the problems addressed and solved in the prior art, and the teachings of nnet. Specific examples of such references are cited and explained in these invalidity contentions, including the cover pleading and charts C1–C15, and Appendix C.