# Backpropagation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overview</strong></td>
<td>5-2</td>
</tr>
<tr>
<td><strong>Fundamentals</strong></td>
<td>5-3</td>
</tr>
<tr>
<td>Architecture</td>
<td>5-3</td>
</tr>
<tr>
<td>Simulation (SIM)</td>
<td>5-8</td>
</tr>
<tr>
<td>Training</td>
<td>5-8</td>
</tr>
<tr>
<td><strong>Faster Training</strong></td>
<td>5-16</td>
</tr>
<tr>
<td>Variable Learning Rate (TRAINDA, TRAINDX)</td>
<td>5-16</td>
</tr>
<tr>
<td>Resilient Backpropagation (TRAINRP)</td>
<td>5-18</td>
</tr>
<tr>
<td>Conjugate Gradient Algorithms</td>
<td>5-20</td>
</tr>
<tr>
<td>Line Search Routines</td>
<td>5-26</td>
</tr>
<tr>
<td>Quasi-Newton Algorithms</td>
<td>5-29</td>
</tr>
<tr>
<td>Levenberg-Marquardt (TRAINLM)</td>
<td>5-31</td>
</tr>
<tr>
<td>Reduced Memory Levenberg-Marquardt (TRAINLM)</td>
<td>5-33</td>
</tr>
<tr>
<td><strong>Speed and Memory Comparison</strong></td>
<td>5-35</td>
</tr>
<tr>
<td><strong>Improving Generalization</strong></td>
<td>5-37</td>
</tr>
<tr>
<td>Regularization</td>
<td>5-38</td>
</tr>
<tr>
<td>Early Stopping</td>
<td>5-41</td>
</tr>
<tr>
<td><strong>Preprocessing and Postprocessing</strong></td>
<td>5-44</td>
</tr>
<tr>
<td>Min and Max (PREMNMX, POSTMNMX, TRAMNMX)</td>
<td>5-44</td>
</tr>
<tr>
<td>Mean and Stand. Dev. (PRESTD, POSTSTD, TRASTD)</td>
<td>5-45</td>
</tr>
<tr>
<td>Principal Component Analysis (PREPCA, TRAPCA)</td>
<td>5-46</td>
</tr>
<tr>
<td>Post-training Analysis (POSTREG)</td>
<td>5-47</td>
</tr>
<tr>
<td><strong>Sample Training Session</strong></td>
<td>5-49</td>
</tr>
<tr>
<td><strong>Limitations and Cautions</strong></td>
<td>5-54</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>5-56</td>
</tr>
</tbody>
</table>
Overview

Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding output vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.

Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule. The term backpropagation refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm which are based on other standard optimization techniques, such as conjugate gradient and Newton methods. The Neural Network Toolbox implements a number of these variations. This chapter will explain how to use each of these routines and will discuss the advantages and disadvantages of each.

Properly-trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input will lead to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs. There are two features of the Neural Network Toolbox which are designed to improve network generalization - regularization and early stopping. These features and their use will be discussed later in this chapter.

This chapter will also discuss preprocessing and postprocessing techniques which can improve the efficiency of network training.

Fundamentals

Architecture

In this section we want to present the architecture of the network which is most commonly used with the backpropagation algorithm - the multilayer feedforward network. The routines in the Neural Network Toolbox can be used to train more general networks, some of these will be briefly discussed in later chapters.

Neuron Model (TAN SIG, LOG SIG, PURELIN)

An elementary neuron with \( R \) inputs is shown below. Each input is weighted with an appropriate \( w \). The sum of the weighted inputs and the bias forms the input to the transfer function \( f \). Neurons may use any differentiable transfer function \( f \) to generate their output.

\[
\begin{align*}
\text{Input} & \quad \text{General Neuron} \\
\sum_{i=1}^{R} w_i p_i & \quad f(\mathbf{Wp} + b) \\
1 & \quad \text{Where...}
\end{align*}
\]

Where...

\( R = \# \text{ Elements in input vector} \)

Multilayer networks often use the log-sigmoid transfer function \( \text{logsig} \).

\[
a = \text{logsig}(n)
\]

Log-Sigmoid Transfer Function
The function \texttt{logsig} generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks may use the tan-sigmoid transfer function \texttt{tansig}.

\begin{align*}
a = \text{tansig}(n)
\end{align*}

Tan-Sigmoid Transfer Function

Occasionally, the linear transfer function \texttt{purelin} is used in backpropagation networks.

\begin{align*}
a = \text{purelin}(n)
\end{align*}

Linear Transfer Function

If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

In backpropagation it is important to be able to calculate the derivatives of any transfer functions used. Each of the transfer functions above, \texttt{tansig, logsig}, and \texttt{purelin}, have a corresponding derivative function: \texttt{dtansig}, \texttt{dlogsig} and \texttt{dpurelin}. To get the name of a transfer function's associated derivative function, call the transfer function with the string 'deriv'.

\begin{verbatim}
tansig('deriv')
ans = dtansig
\end{verbatim}
The three transfer functions described here are the most commonly used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired. See Chapter 11, “Advanced Topics” for more information.

**Feedforward Network**
A single-layer network of $S$ logsig neurons having $R$ inputs is shown below in full detail on the left and with a layer diagram on the right.

Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range $-1$ to $+1$.

On the other hand, if it is desirable to constrain the outputs of a network (such as between 0 and 1) then the output layer should use a sigmoid transfer function (such as logsig).

As noted in Chapter 2, for multiple-layer networks we use the number of the layers to determine the superscript on the weight matrices. The appropriate notation is used in the two-layer \texttt{tansig/purelin} network shown next.
This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities, arbitrarily well, given sufficient neurons in the hidden layer.

**Creating a Network (NEWFF).** The first step in training a feedforward network is to create the network object. The function `newff` creates a trainable feedforward network. It requires four inputs and returns the network object. The first input is an R by 2 matrix of minimum and maximum values for each of the R elements of the input vector. The second input is an array containing the sizes of each layer. The third input is a cell array containing the names of the transfer functions to be used in each layer. The final input contains the name of the training function to be used.

For example, the following command will create a two-layer network. There will be one input vector with two elements, three neurons in the first layer and one neuron in the second (output) layer. The transfer function in the first layer will be tan-sigmoid, and the output layer transfer function will be linear. The values for the first element of the input vector will range between -1 and 2, the values of the second element of the input vector will range between 0 and 5, and the training function will be `traingd` (which will be described in a later section).

```matlab
net = newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingd');
```

This command creates the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you may wish to re-initialize the weights, or to perform a custom initialization. The next section explains the details of the initialization process.
Initializing Weights (INIT, INITNW, RANDS). Before training a feedforward network, the weights and biases must be initialized. The initial weights and biases are created with the command init. This function takes a network object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized:

```matlab
net = init(net);
```

The specific technique which is used to initialize a given network will depend on how the network parameters `net.initFcn` and `net.layers{i}.initFcn` are set. The parameter `net.initFcn` is used to determine the overall initialization function for the network. The default initialization function for the feedforward network is `initlay`, which allows each layer to use its own initialization function. With this setting for `net.initFcn`, the parameters `net.layers{i}.initFcn` are used to determine the initialization method for each layer.

For feedforward networks there are two different layer initialization methods which are normally used: `initwb` and `initnw`. The `initwb` function causes the initialization to revert to the individual initialization parameters for each weight matrix (`net.inputWeights{i,j}.initFcn`) and bias. For the feedforward networks the weight initialization is usually set to `rands`, which sets weights to random values between -1 and 1. It is normally used when the layer transfer function is linear.

The function `initnw` is normally used for layers of feedforward networks where the transfer function is sigmoid. It is based on the technique of Nguyen and Widrow [NgWi90] and generates initial weight and bias values for a layer so that the active regions of the layer's neurons will be distributed roughly evenly over the input space. It has several advantages over purely random weights and biases: (1) few neurons are wasted (since the active regions of all the neurons are in the input space), (2) training works faster (since each area of the input space has active neuron regions).

The initialization function `init` is called by `newff`, therefore the network is automatically initialized with the default parameters when it is created, and `init` does not have to be called separately. However, the user may want to re-initialize the weights and biases, or to use a specific method of initialization.

For example, in the network that we just created, using `newff`, the default initialization for the first layer would be `initnw`. If we wanted to re-initialize
the weights and biases in the first layer using the `rands` function, we would issue the following commands:

```matlab
net.layers{1}.initFcn = 'initwb';
net.inputWeights{1,1}.initFcn = 'rands';
net.biases{1,1}.initFcn = 'rands';
net.biases{2,1}.initFcn = 'rands';
net = init(net);
```

**Simulation (SIM)**

The function `sim` simulates a network. `sim` takes the network input `p`, and the network object `net`, and returns the network outputs `a`. Here is how `sim` can be used to simulate the network we created above for a single input vector:

```matlab
p = [1;2];
a = sim(net,p)
a =
-0.1011
```

(If you try these commands, your output may be different, depending on the state of your random number generator when the network was initialized.) Below, `sim` is called to calculate the outputs for a concurrent set of three input vectors.

```matlab
p = [1 3 2;2 4 1];
a = sim(net,p)
a =
-0.1011   -0.2308   0.4955
```

**Training**

Once the network weights and biases have been initialized, the network is ready for training. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. The training process requires a set of examples of proper network behavior - network inputs `p` and target outputs `t`. During training the weights and biases of the network are iteratively adjusted to minimize the network performance function `net.performFcn`. The default performance function for feedforward networks is mean square error `mse` - the average squared error between the network outputs `a` and the target outputs `t`. 
The remainder of this chapter will describe several different training algorithms for feedforward networks. All of these algorithms use the gradient of the performance function to determine how to adjust the weights to minimize performance. The gradient is determined using a technique called backpropagation, which involves performing computations backwards through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapter 11 of [HDB96]. The basic backpropagation training algorithm, in which the weights are moved in the direction of the negative gradient, is described in the next section. Later sections will describe more complex algorithms that increase the speed of convergence.

**Backpropagation Algorithm**

There are many variations of the backpropagation algorithm, several of which will be discussed in this chapter. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly - the negative of the gradient. One iteration of this algorithm can be written

\[ x_{k+1} = x_k - \alpha_k g_k, \]

where \( x_k \) is a vector of current weights and biases, \( g_k \) is the current gradient, and \( \alpha_k \) is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: incremental mode and batch mode. In the incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In the batch mode all of the inputs are applied to the network before the weights are updated. The next section will describe the incremental training, and the following section will describe batch training.

**Incremental Training (ADAPT)**

The function `adapt` is used to train networks in the incremental mode. This function takes the network object and the inputs and the targets from the training set, and returns the trained network object and the outputs and errors of the network for the final weights and biases.

There are several network parameters which must be set in order to guide the incremental training. The first is `net.adaptFcn`, which determines which incremental mode training function is to be used. The default for feedforward
networks is adaptwb, which allows each weight and bias to assign its own function. These individual learning functions for the weights and biases are set by the parameters net.biases{i,j}.learnFcn, net.inputWeights{i,j}.learnFcn, and net.layerWeights{i,j}.learnFcn.

Gradient Descent (LEARDGD). For the basic steepest (gradient) descent algorithm, the weights and biases are moved in the direction of the negative gradient of the performance function. For this algorithm, the individual learning function parameters for the weights and biases are set to 'learngd'. The following commands illustrate how these parameters are set for the feedforward network we created earlier.

```matlab
net.biases{1,1}.learnFcn = 'learngd';
net.biases{2,1}.learnFcn = 'learngd';
net.layerWeights{2,1}.learnFcn = 'learngd';
net.inputWeights{1,1}.learnFcn = 'learngd';
```

The function learngd has one learning parameter associated with it - the learning rate lr. The changes to the weights and biases of the network are obtained by multiplying the learning rate times the negative of the gradient.

The larger the learning rate, the bigger the step. If the learning rate is made too large the algorithm will become unstable. If the learning rate is set too small, the algorithm will take a long time to converge. See page 12-8 of [HDB96] for a discussion of the choice of learning rate.

The learning rate parameter is set to the default value for each weight and bias when the learnFcn is set to learngd, as in the code above, although you can change its value if you desire. The following command demonstrates how you can set the learning rate to 0.2 for the layer weights. The learning rate can be set separately for each weight and bias.

```matlab
net.layerWeights{2,1}.learnParam.lr = 0.2;
```

The final parameter to be set for sequential training is net.adaptParam.passes, which determines the number of passes through the training set during training. Here we set the number of passes to 200.

```matlab
net.adaptParam.passes = 200;
```
We are now almost ready to train the network. It remains to set up the training set. Here is a simple set of inputs and targets which we will use to illustrate the training procedure:

\[ p = \begin{bmatrix} -1 & 1 & 2 & 2; 0 & 5 & 0 & 5 \end{bmatrix}; \]
\[ t = \begin{bmatrix} -1 & 1 & 1 \end{bmatrix}; \]

If we want the learning algorithm to update the weights after each input pattern is presented, we need to convert the matrices of inputs and targets into cell arrays, with a cell for each input vector and target:

\[ p = \text{num2cell}(p,1); \]
\[ t = \text{num2cell}(t,1); \]

We are now ready to perform the incremental training using the adapt function:

\[ [\text{net}, a, e] = \text{adapt}([\text{net}, p, t]); \]

After the training is complete we can simulate the network to test the quality of the training.

\[ a = \text{sim}([\text{net}, p]) \]
\[ a = \begin{bmatrix} -0.9995 & -1.0000 & 1.0001 & 1.0000 \end{bmatrix} \]

**Gradient Descent With Momentum (LEARDGDM).** In addition to learnfgd, there is another incremental learning algorithm for feedforward networks that often provides faster convergence - learnfgd, steepest descent with momentum. Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a low pass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12-9 of [HDB96] for a discussion of momentum.

Momentum can be added to backpropagation learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the backpropagation rule. The magnitude of the effect that the last weight change is allowed to have is mediated by a momentum constant, \( mc \), which can be any number between 0 and 1. When the momentum constant is 0 a weight change is based solely on the gradient. When the
momentum constant is 1 the new weight change is set to equal the last weight change and the gradient is simply ignored.

The \texttt{learngdm} function is invoked using the same steps shown above for the \texttt{learngd} function, except that both the \texttt{mc} and \texttt{lr} learning parameters can be set. Different parameter values can be used for each weight and bias, since each weight and bias has its own learning parameters.

The following commands will cause the previously created network to be incrementally trained using \texttt{learngdm} with the default learning parameters.

\begin{verbatim}
net.biases{1,1}.learnFcn = 'learngdm';
net.biases{2,1}.learnFcn = 'learngdm';
net.layerWeights{2,1}.learnFcn = 'learngdm';
net.inputWeights{1,1}.learnFcn = 'learngdm';
[net,a,e]=adapt(net,p,t);
\end{verbatim}

\textbf{Batch Training (TRAIN).} The alternative to incremental training is batch training, which is invoked using the function \texttt{train}. In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in the weights and biases. For a discussion of batch training with the backpropagation algorithm see page 12-7 of [HDB96].

\textbf{Batch Gradient Descent (TRAINGD).} The batching equivalent of the incremental function \texttt{learngd} is \texttt{traingd}, which implements the batching form of the standard steepest descent training function. The weights and biases are updated in the direction of the negative gradient of the performance function. If you wish to train a network using batch steepest descent, you should set the \texttt{network} \texttt{trainFcn} to \texttt{traingd} and then call the function \texttt{train}. Unlike the learning functions in the previous section, which were assigned separately to each weight matrix and bias vector in the network, there is only one training function associated with a given network.

There are seven training parameters associated with \texttt{traingd:epochs, show, goal, time, min_grad, max_fail, and lr}. The learning rate \texttt{lr} has the same meaning here as it did for \texttt{learngd} - it is multiplied times the negative of the gradient to determine the changes to the weights and biases. The training status will be displayed every \texttt{show} iterations of the algorithm. The other parameters determine when the training is stopped. The training will stop if the number of iterations exceeds \texttt{epochs}, if the performance function drops...
below \texttt{goal}, if the magnitude of the gradient is less than \texttt{mingrad}, or if the training time is longer than \texttt{time} seconds. We will discuss \texttt{max_fail}, which is associated with the early stopping technique, in the section on improving generalization.

The following code will recreate our earlier network, and then train it using batch steepest descent. (Note that for batch training all of the inputs in the training set are placed in one matrix.)

```matlab
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingd');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=train(net,p,t);
```

```
TRAINGD, Epoch 0/300, MSE 1.59423/1e-05, Gradient 2.76799/1e-10
TRAINGD, Epoch 50/300, MSE 0.00236382/1e-05, Gradient 0.0495292/1e-10
TRAINGD, Epoch 100/300, MSE 0.000435947/1e-05, Gradient 0.0161202/1e-10
TRAINGD, Epoch 150/300, MSE 8.68462e-05/1e-05, Gradient 0.00769588/1e-10
TRAINGD, Epoch 200/300, MSE 1.45042e-05/1e-05, Gradient 0.00325667/1e-10
TRAINGD, Epoch 211/300, MSE 9.64816e-06/1e-05, Gradient 0.00266775/1e-10
TRAINGD, Performance goal met.
```

```matlab
a = sim(net,p)
a =
-1.0010  -0.9989  1.0018  0.9985
```

Try the Neural Network Design Demonstration \texttt{nnd12sd1[HDB96]} for an illustration of the performance of the batch gradient descent algorithm.

**Batch Gradient Descent With Momentum (TRAIN GDM).** The batch form of gradient descent with momentum is invoked using the training function \texttt{train gdm}. This algorithm is equivalent to \texttt{learngd}, with two exceptions. First, the gradient is computed by summing the gradients calculated at each training example, and
the weights and biases are only updated after all training examples have been presented. Second, if the new performance function on a given iteration exceeds the performance function on a previous iteration by more than a predefined ratio max_perf_inc (typically 1.04), the new weights and biases are discarded, and the momentum coefficient mc is set to zero.

In the following code we recreate our previous network and retrain it using gradient descent with momentum. The training parameters for traingdm are the same as those for traingd, with the addition of the momentum factor mc and the maximum performance increase max_perf_inc. (The training parameters are reset to the default values whenever net.trainFcn is set to traingdm.)

```matlab
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingdm');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net=train(net,p,t);

TRAINGDM, Epoch 0/300, MSE 3.6913/1e-05, Gradient 4.54729/1e-10
TRAINGDM, Epoch 50/300, MSE 0.00532188/1e-05, Gradient 0.213222/1e-10
TRAINGDM, Epoch 100/300, MSE 0.348868e-05/1e-05, Gradient 0.0409749/1e-10
TRAINGDM, Epoch 114/300, MSE 0.062355e-06/1e-05, Gradient 0.00908756/1e-10
TRAINGDM, Performance goal met.
```

```matlab
a = sim(net,p)
a =
   -1.0026   -1.0044    0.9969    0.9992
```
Note that since we re-initialized the weights and biases before training, we obtain a different mean square error than we did using `traingd`. If we were to re-initialize and train again using `traingdm`, we would get yet a different mean square error. The random choice of initial weights and biases will affect the performance of the algorithm. If you wish to compare the performance of different algorithms, you should test each using several different sets of initial weights and biases.

Try the Neural Network Design Demonstration `nnd12mo` [HDB96] for an illustration of the performance of the batch momentum algorithm.
Faster Training

The previous section presented two backpropagation training algorithms: gradient descent and gradient descent with momentum. These two methods are often too slow for practical problems. In this section we will discuss several high performance algorithms which can converge from ten to one hundred times faster than the algorithms discussed previously. All of the algorithms in this section operate in the batch mode and are invoked using `train`.

These faster algorithms fall into two main categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm. One heuristic modification is the momentum technique, which was presented in the previous section. This section will discuss two more heuristic techniques: variable learning rate backpropagation, `traingda`, and resilient backpropagation `trainrp`.

The second category of fast algorithms uses standard numerical optimization techniques. (See Chapter 9 of [HDB96] for a review of basic numerical optimization.) Later in this section we will present three types of numerical optimization techniques for neural network training: conjugate gradient (`traincgp`, `traincgf`, `traincgb`, `trainscg`), quasi-Newton (`trainbfg`, `trainoss`), and Levenberg-Marquardt (`trainlm`).

Variable Learning Rate (TRAINGDA, TRAINGDX)

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm may oscillate and become unstable. If the learning rate is too small, the algorithm will take too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

The performance of the steepest descent algorithm can be improved if we allow the learning rate to change during the training process. An adaptive learning rate will attempt to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.
An adaptive learning rate requires some changes in the training procedure used by \textit{traind}. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio $\text{ratio}_{\text{max perf inc}}$ (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by $\text{lr}_{\text{dec}} = 0.7$). Otherwise the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by $\text{lr}_{\text{inc}} = 1.05$).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus a near optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it gets decreased until stable learning resumes.

Try the Neural Network Design Demonstration \texttt{nnd12vl} [HDB96] for an illustration of the performance of the variable learning rate algorithm.
Backpropagation training with an adaptive learning rate is implemented with the function `traingda` which is called just like `traingd` except for the additional training parameters `max_perf_inc`, `lr_dec`, and `lr_inc`. Here is how it is called to train our previous two-layer network:

```matlab
net = newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingda');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=train(net,p,t);
```

```
TRAINGDA, Epoch 0/300, MSE 1.71149/1e-05, Gradient 2.6397/1e-06
TRAINGDA, Epoch 44/300, MSE 7.47952e-06/1e-05, Gradient 0.00251265/1e-06
TRAINGDA, Performance goal met.
a = sim(net,p)
a =
    1.0036   -0.9960    1.0008    0.9991
```

The function `traingdx` combines adaptive learning rate with momentum training. It is invoked in the same way as `traingda`, except that it has the momentum coefficient `mc` as an additional training parameter.

**Resilient Backpropagation (TRAINRP)**

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called squashing functions, since they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slope must approach zero as the input gets large. This causes a problem when using steepest descent to train a multilayer network with sigmoid functions, since the gradient can have a very small magnitude, and therefore cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative is used to determine the direction of the weight
update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor $delt\_inc$ whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor $delt\_dec$ whenever the derivative with respect that weight changes sign from the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating the weight change will be reduced. If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change will be increased. A complete description of the Rprop algorithm is given in [ReBr93].

In the following code we recreate our previous network and train it using the Rprop algorithm. The training parameters for `trainrp` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `delt\_inc`, `delt\_dec`, `delta0`, `deltamax`. We have previously discussed the first eight parameters. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below we leave most of the training parameters at the default values. We do reduce `show` below our previous value, because Rprop generally converges much faster than the previous algorithms.

```matlab
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'trainrp');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net=train(net,p,t);
TRAINRP, Epoch 0/300, MSE 0.469151/1e-05, Gradient 1.4258/1e-06
TRAINRP, Epoch 10/300, MSE 0.000789506/1e-05, Gradient 0.0554529/1e-06
TRAINRP, Epoch 20/300, MSE 7.13065e-06/1e-05, Gradient 0.00346986/1e-06
TRAINRP, Performance goal met.
a = sim(net,p)
a =
-1.0026 -0.9963 0.9978 1.0017
```
Rprop is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. We do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

**Conjugate Gradient Algorithms**

The basic backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient). This is the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In this section we will present four different variations of conjugate gradient algorithms.


In most of the training algorithms that we have discussed up to this point, a learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size which will minimize the performance function along that line. There are five different search functions that are included in the toolbox, and these will be discussed at the end of this section. Any of these search functions can be used interchangeably with a variety of the training functions described in the remainder of this chapter. Some search functions are best suited to certain training functions, although the optimum choice can vary according to the specific application. An appropriate default search function is assigned to each training function, but this can be modified by the user.

**Fletcher-Reeves Update (TRAINCGF)**

All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

\[ p_0 = -g_0 \]
A line search is then performed to determine the optimal distance to move along the current search direction:

\[ x_{k+1} = x_k + \alpha_k p_k \]

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

\[ p_k = -g_k + \beta_k p_{k-1} \]

The various versions of conjugate gradient are distinguished by the manner in which the constant \( \beta_k \) is computed. For the Fletcher-Reeves update the procedure is

\[ \beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}. \]

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

In the following code we re-initialize our previous network and retrain it using the Fletcher-Reeves version of the conjugate gradient algorithm. The training parameters for \texttt{traincgf} are epochs, show, goal, time, min_grad, max_fail, srchFcn, scal_tol, alpha, beta, delta, gama, low_lim, up_lim, maxstep, minstep, bmax. We have previously discussed the first six parameters. The parameter \texttt{srchFcn} is the name of the line search function. It can be any of the functions described later in this section (or a user-supplied function). The remaining parameters are associated with specific line search routines and are described later in this section. The default line search routines \texttt{srchcha} is used in this example. \texttt{traincgf} generally converges in fewer iterations than \texttt{trainrp} (although there is more computation required in each iteration).
net = newff([-1 2; 0 5], [3, 1], {'tansig', 'purelin'}, 'traincrgf');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = train(net, p, t);
TRAINCGR-srchcha, Epoch 0/300, MSE 2.15911/1e-05, Gradient 3.17681/1e-06
TRAINCGR-srchcha, Epoch 5/300, MSE 0.111081/1e-05, Gradient 0.602109/1e-06
TRAINCGR-srchcha, Epoch 10/300, MSE 0.0095015/1e-05, Gradient 0.197436/1e-06
TRAINCGR-srchcha, Epoch 15/300, MSE 0.000508668/1e-05,
Gradient 0.0439273/1e-06
TRAINCGR-srchcha, Epoch 17/300, MSE 1.33611e-06/1e-05,
Gradient 0.00562836/1e-06
TRAINCGR. Performance goal met.
a = sim(net, p)
a =
-1.0001  -1.0023  0.9999  1.0002

The conjugate gradient algorithms are usually much faster than
variable learning rate backpropagation, and are sometimes faster than trainrp, although the results will vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms, so they are often a good choice for networks with a large number of weights.

Try the Neural Network Design Demonstration nnd12cg [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

Polak-Ribiére Update (TRAINCGP)

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, the search direction at each iteration is determined by

\[ p_k = -g_k + \beta_k p_{k-1} \]
For the Polak-Ribiére update, the constant $\beta_k$ is computed by

$$
\beta_k = \frac{\Delta g_{k-1}^T g_k}{g_k^T g_{k-1}}.
$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FIRE64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

In the following code we recreate our previous network and train it using the Polak-Ribiére version of the conjugate gradient algorithm. The training parameters for `traincgp` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epoch` are set to the same values as they were for `traincgf`.

```matlab
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traincgp');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=train(net,p,t);
TRAINCGP-srchcha, Epoch 0/300, MSE 1.21966/1e-05, Gradient 1.77008/1e-06
TRAINCGP-srchcha, Epoch 5/300, MSE 0.227447/1e-05, Gradient 0.86507/1e-06
TRAINCGP-srchcha, Epoch 10/300, MSE 0.000237395/1e-05, Gradient 0.0174276/1e-06
TRAINCGP-srchcha, Epoch 15/300, MSE 9.28243e-05/1e-05, Gradient 0.00485746/1e-06
TRAINCGP-srchcha, Epoch 20/300, MSE 1.46146e-05/1e-05, Gradient 0.000912838/1e-06
TRAINCGP-srchcha, Epoch 25/300, MSE 1.05893e-05/1e-05, Gradient 0.00238173/1e-06
TRAINCGP-srchcha, Epoch 26/300, MSE 9.10561e-06/1e-05, Gradient 0.00197441/1e-06
TRAINCGP-srchcha, Epoch 26/300, MSE 9.10561e-06/1e-05, Gradient 0.00197441/1e-06
TRAINCGP, Performance goal met.
a = sim(net,p)
a =
-0.9967  -1.0018   0.9958  1.0022
The `traincgp` routine has performance similar to `traincgf`. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

**Powell-Beale Restarts (TRAINCGB)**

For all conjugate gradient algorithms, the search direction will be periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods which can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. For this technique we will restart if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality:

\[ \| g_k - 1 g_{k-1} \| \geq 0.2 \| g_k \|^2. \]

If this condition is satisfied, the search direction is reset to the negative of the gradient.

In the following code we recreate our previous network and train it using the Powell-Beale version of the conjugate gradient algorithm. The training parameters for `traincgb` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epoch` are set to the same values as they were for `traincgf`.

```matlab
net = newff([-1 2; 0 5], [3, 1], {'tansig', 'purelin'}, 'traincgb');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = train(net, p, t);
```

Training details:

- TRAINCGB-srchcha, Epoch 0/300, MSE 2.5245/1e-05, Gradient 3.66882/1e-06
- TRAINCGB-srchcha, Epoch 5/300, MSE 4.86255e-07/1e-05, Gradient 0.00145878/1e-06
- TRAINCGB, Performance goal met.

```
a = sim(net, p)
a =
  -0.9997   -0.9998    1.0000    1.0014
```
The `traincgb` routine has performance which is somewhat better than `traincgp` for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

**Scaled Conjugate Gradient (TRAINSCG)**

Each of the conjugate gradient algorithms which we have discussed so far require a line search at each iteration. This line search is computationally expensive, since it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller [Moll93], was designed to avoid the time consuming line search. This algorithm is too complex to explain in a few lines, but the basic idea is to combine the model-trust region approach, which is used in the Levenberg-Marquardt algorithm described later, with the conjugate gradient approach. See [Moll93] for a detailed explanation of the algorithm.

In the following code we re-initialize our previous network and retrain it using the scaled conjugate gradient algorithm. The training parameters for `trainscg` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `sigma`, `lambda`. We have previously discussed the first six parameters. The parameter `sigma` determines the change in the weight for the second derivative approximation. The parameter `lambda` regulates the indefiniteness of the Hessian. The parameters `show` and `epoch` are set to 10 and 300, respectively.

```matlab
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'trainscg');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
net.trainParam.time = 300;
net.trainParam.min_grad = 1e-5;
net.trainParam.max_fail = 1e-5;
net=train(net,p,t);
```

<table>
<thead>
<tr>
<th>Epoch</th>
<th>MSE (1e-05)</th>
<th>Gradient (1e-06)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.17697</td>
<td>0.00673703</td>
</tr>
<tr>
<td>300</td>
<td>9.38923</td>
<td>0.0049926</td>
</tr>
</tbody>
</table>

```
a = sim(net,p)
a =
-1.0057   -1.0008    1.0019    1.0005
```
The `trainscg` routine may require more iterations to converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed. The storage requirements for the scaled conjugate gradient algorithm are about the same as those of Fletcher-Reeves.

**Line Search Routines**

Several of the conjugate gradient and quasi-Newton algorithms require that a line search be performed. In this section we describe five different line searches which can be used. In order to use any of these search routines you simply set the training parameter `srchfcn` equal to the name of the desired search function, as has been described in previous sections. It is often difficult to predict which of these routines will provide the best results for any given problem, but we have set the default search function to an appropriate initial choice for each training function, so you may never need to modify this parameter.

**Golden Section Search (SRCHGOL)**

The golden section search `srchgol` is a linear search which does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of `delta` and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determines a section of the interval which can be discarded, and a new interior point is placed within the new interval. This procedure is continued until the interval of uncertainty is reduced to a width of `tol`, which is equal to `delta/scale_tol`.

See [HDB96], starting on page 12-16, for a complete description of the golden section search. Try the Neural Network Design Demonstration `nnd12sd1` [HDB96] for an illustration of the performance of the golden section search in combination with a conjugate gradient algorithm.
Brent's Search (SRCHBRE)

Brent's search is a linear search which is a hybrid combination of the golden section search and a quadratic interpolation. Function comparison methods, like the golden section search, have a first-order rate of convergence, while polynomial interpolation methods have an asymptotic rate which is faster than superlinear. On the other hand, the rate of convergence for the golden section search starts when the algorithm is initialized, whereas the asymptotic behavior for the polynomial interpolation methods may take many iterations to become apparent. Brent's search attempts to combine the best features of both approaches.

For Brent's search we begin with the same interval of uncertainty that we used with the golden section search, but some additional points are computed. A quadratic function is then fitted to these points and the minimum of the quadratic function is computed. If this minimum is within the appropriate interval of uncertainty, it is used in the next stage of the search and a new quadratic approximation is performed. If the minimum falls outside the known interval of uncertainty, then a step of the golden section search is performed.

See [Bren73] for a complete description of this algorithm. This algorithm has the advantage that it does not require computation of the derivative. The derivative computation requires a backpropagation through the network, which involves more computation than a forward pass. However, the algorithm may require more performance evaluations than algorithms which use derivative information.

Hybrid Bisection-Cubic Search (SRCHHYB)

Like Brent's search, srchhyb is a hybrid algorithm. It is a combination of bisection and cubic interpolation. For the bisection algorithm, one point is located in the interval of uncertainty and the performance and its derivative are computed. Based on this information, half of the interval of uncertainty is discarded. In the hybrid algorithm, a cubic interpolation of the function is obtained by using the value of the performance and its derivative at the two end points. If the minimum of the cubic interpolation falls within the known interval of uncertainty, then it is used to reduce the interval of uncertainty. Otherwise, a step of the bisection algorithm is used.

See [Scal85] for a complete description of the hybrid bisection-cubic search. This algorithm does require derivative information, so it performs more computations at each step of the algorithm than the golden section search or Brent's algorithm.
Charalambous' Search (SRCHCHA)

The method of Charalambous [SRCHCHA] was designed to be used in combination with a conjugate gradient algorithm for neural network training. Like the previous two methods, it is a hybrid search. It uses a cubic interpolation, together with a type of sectioning.

See [Char92] for a description of Charalambous’ search. We have used this routine as the default search for most of the conjugate gradient algorithms, since it appears to produce excellent results for many different problems. It does require the computation of the derivatives (backpropagation) in addition to the computation of performance, but it overcomes this limitation by locating the minimum with fewer steps. This is not true for all problems, and you may want to experiment with other line searches.

Backtracking (SRCHBAC)

The backtracking search routine [SRCHBAC] is best suited to use with the quasi-Newton optimization algorithms. It begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained. On the first step it uses the value of performance at the current point and at a step multiplier of 1, and also the derivative of performance at the current point, to obtain a quadratic approximation to the performance function along the search direction. The minimum of the quadratic approximation becomes a tentative optimum point (under certain conditions) and the performance at this point is tested. If the performance is not sufficiently reduced, a cubic interpolation is obtained and the minimum of the cubic interpolation becomes the new tentative optimum point. This process is continued until a sufficient reduction in the performance is obtained.

The backtracking algorithm is described in [DeSc83]. It was used as the default line search for the quasi-Newton algorithms, although it may not be the best technique for all problems.
Quasi-Newton Algorithms

BFGS Algorithm (TRAIN BFG)

Newton’s method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton’s method is

\[ x_{k+1} = x_k - A_k^{-1} g_k, \]

where \( A_k \) is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton’s method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that are based on Newton’s method but which don’t require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method which has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm has been implemented in the \textit{trainbfg} routine.

In the following code we re-initialize our previous network and retrain it using the BFGS quasi-Newton algorithm. The training parameters for \textit{trainbfg} are the same as those for \textit{traincgf}. The default line search routine \textit{srchbac} is used in this example. The parameters \texttt{show} and \texttt{epoch} are set to 5 and 300, respectively.
net = newff([-1, 2; 0, 5], [3, 1], {'tansig', 'purelin'}, 'trainbfg');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2; 0 5 0 5];
t = [-1 -1 1 1];
net = train(net, p, t);

TRAINBFG-srchbac, Epoch 0/300, MSE 0.492231/1e-05, Gradient 2.16307/1e-06
TRAINBFG-srchbac, Epoch 5/300, MSE 0.000744953/1e-05, Gradient 0.0196826/1e-06
TRAINBFG-srchbac, Epoch 8/300, MSE 7.69867e-06/1e-05, Gradient 0.00497404/1e-06
TRAINBFG, Performance goal met.
a = sim(net, p)
a =
0.9995  -1.0004  1.0008  0.9945

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is $n \times n$, where $n$ is equal to the number of weights and biases in the network. For very large networks it may be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, trainbfg can be an efficient training function.

One Step Secant Algorithm (TRAINOSS)

Since the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

In the following code we re-initialize our previous network and retrain it using the one step secant algorithm. The training parameters for trainoss are the same as those for traincfg. The default line search routine srchbac is used in
this example. The parameters show and epoch are set to 10 and 300, respectively.

```matlab
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin','trainoss'});
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 1 1 1];
net=train(net,p,t);
TRAINOSS-srchbac, Epoch 0/300, MSE 0.665136/1e-05, Gradient
1.61966/1e-06
TRAINOSS-srchbac, Epoch 5/300, MSE 0.000321921/1e-05, Gradient
0.0261425/1e-06
TRAINOSS-srchbac, Epoch 7/300, MSE 7.85697e-06/1e-05, Gradient
0.00527342/1e-06
TRAINOSS, Performance goal met.
```

The one step secant method is described in [Batt92]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

**Levenberg-Marquardt (TRAINLM)**

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

\[ H = J^T J, \]

and the gradient can be computed as

\[ g = J^T e \]
where $J$ is the Jacobian matrix, which contains first derivatives of the network errors with respect to the weights and biases, and $e$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$x_{k+1} = x_k - [JJ^T + \mu I]^{-1}J^Te.$$

When the scalar $\mu$ is zero, this is just Newton’s method, using the approximate Hessian matrix. When $\mu$ is large, this becomes gradient descent with a small step size. Newton’s method is faster and more accurate near an error minimum, so the aim is to shift towards Newton’s method as quickly as possible. Thus, $\mu$ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function will always be reduced at each iteration of the algorithm.

In the following code we re-initialize our previous network and retrain it using the Levenberg-Marquardt algorithm. The training parameters for trainlm are epochs, show, goal, time, min_grad, max_fail, mu, mu_dec, mu_inc, mu_max, and mem_reduc. We have discussed the first six parameters earlier. The parameter $\mu$ is the initial value for $\mu$. This value is multiplied by $\mu_{\text{dec}}$ whenever the performance function is reduced by a step. It is multiplied by $\mu_{\text{inc}}$ whenever a step would increase the performance function. If $\mu$ becomes larger than $\mu_{\text{max}}$, the algorithm is stopped. The parameter $\text{mem}_{\text{reduc}}$ is used to control...
the amount of memory used by the algorithm. It will be discussed in the next section. The parameters show and epoch are set to 5 and 300, respectively.

```matlab
net = newff([-1 2; 0 5], [3,1], {'tansig', 'purelin'}, 'trainlm');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = train(net, p, t);
```

```
TRAINLM, Epoch 0/300, MSE 2.7808/1e-05, Gradient 7.77931/1e-10
TRAINLM, Epoch 4/300, MSE 3.67935e-08/1e-05, Gradient 0.000808272/1e-10
TRAINLM, Performance goal met.
a = sim(net, p)
a =
   -1.0000   -1.0000    1.0000    0.9996
```

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has a very efficient MATLAB implementation, since the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB setting.

Try the Neural Network Design Demonstration nnd12m [HDB96] for an illustration of the performance of the batch gradient descent algorithm.

**Reduced Memory Levenberg-Marquardt (TRAINLM)**

The main drawback of the Levenberg-Marquardt algorithm is that it requires the storage of some matrices which can be quite large for certain problems. The size of the Jacobian matrix is $Q \times n$, where $Q$ is the number of training sets and $n$ is the number of weights and biases in the network. It turns out that this matrix does not have to be computed and stored as a whole. For example, if we were to divide the Jacobian into two equal submatrices we could compute the approximate Hessian matrix as follows:
Therefore the full Jacobian does not have to exist at one time. The approximate Hessian can be computed by summing a series of subterms. Once one subterm has been computed, the corresponding submatrix of the Jacobian can be cleared.

When using the training function `trainlm`, the parameter `mem_reduc` is used to determine how many rows of the Jacobian are to be computed in each submatrix. If `mem_reduc` is set to 1, then the full Jacobian is computed, and no memory reduction is achieved. If `mem_reduc` is set to 2, then only half of the Jacobian will be computed at one time. This saves half of the memory used by the calculation of the full Jacobian.

There is a drawback to using memory reduction. A significant computational overhead is associated with computing the Jacobian in submatrices. If you have enough memory available, then it is better to set `mem_reduc` to 1 and to compute the full Jacobian. If you have a large training set, and you are running out of memory, then you should set `mem_reduc` to 2, and try again. If you still run out of memory, continue to increase `mem_reduc`.

Even if you use memory reduction, the Levenberg-Marquardt algorithm will always compute the approximate Hessian matrix, which has dimensions $n \times n$. If your network is very large, then you may run out of memory. If this is the case, then you will want to try `trainoss`, `trainrp`, or one of the conjugate gradient algorithms.

\[
H = J^T J = \begin{bmatrix} J_1^T J_1 & J_1^T J_2 \\ J_2^T J_1 & J_2^T J_2 \end{bmatrix} = J_1^T J_1 + J_2^T J_2.
\]
Speed and Memory Comparison

It is very difficult to know which training algorithm will be the fastest for a given problem. It will depend on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, and the error goal. In general, on networks which contain up to a few hundred weights the Levenberg-Marquardt algorithm will have the fastest convergence. This advantage is especially noticeable if very accurate training is required. The quasi-Newton methods are often the next fastest algorithms on networks of moderate size. The BFGS algorithm does require storage of the approximate Hessian matrix, but is generally faster than the conjugate gradient algorithms. Of the conjugate gradient algorithms, the Powell-Beale procedure requires the most storage, but usually has the fastest convergence. Rprop and the scaled conjugate gradient algorithm do not require a line search and have small storage requirements. They are reasonably fast, and are very useful for large problems. The variable learning rate algorithm is usually much slower than the other methods, and has about the same storage requirements as Rprop, but it can still be useful for some problems. There are certain situations in which it is better to converge more slowly. For example, when using early stopping you may have inconsistent results if you use an algorithm which converges too quickly. You may overshoot the point at which the error on the validation set is minimized.

For most situations, we recommend that you try the Levenberg-Marquardt algorithm first. If this algorithm requires too much memory, then try the BFGS algorithm, or one of the conjugate gradient methods. The Rprop algorithm is also very fast, and has relatively small memory requirements.

The following table gives some example convergence times for the various algorithms on one particular regression problem. In this problem a 1-10-1 network was trained on a data set with 41 input/output pairs until a mean square error performance of 0.01 was obtained. Twenty different test runs were made for each training algorithm on a Macintosh Powerbook 1400 to obtain the average numbers shown in the table. These numbers should be used with caution, since the performances shown here may not be typical for these algorithms on other types of problems.
You may notice that there is not a clear relationship between the number of floating point operations and the time required to reach convergence. This is because some of the algorithms can take advantage of efficient built-in MATLAB functions. This is especially true for the Levenberg-Marquardt algorithm.

<table>
<thead>
<tr>
<th>Function</th>
<th>Technique</th>
<th>Time</th>
<th>Epochs</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>traindx</td>
<td>Variable Learning Rate</td>
<td>57.71</td>
<td>980</td>
<td>2.50</td>
</tr>
<tr>
<td>trainrp</td>
<td>Rprop</td>
<td>12.95</td>
<td>185</td>
<td>0.56</td>
</tr>
<tr>
<td>trainscg</td>
<td>Scaled Conj. Grad.</td>
<td>16.06</td>
<td>106</td>
<td>0.70</td>
</tr>
<tr>
<td>traincgf</td>
<td>Fletcher-Powell CG</td>
<td>16.40</td>
<td>81</td>
<td>0.99</td>
</tr>
<tr>
<td>traincgp</td>
<td>Polak-Ribiére CG</td>
<td>19.16</td>
<td>89</td>
<td>0.75</td>
</tr>
<tr>
<td>traincgb</td>
<td>Powell-Beale CG</td>
<td>15.03</td>
<td>74</td>
<td>0.59</td>
</tr>
<tr>
<td>trainoss</td>
<td>One-Step-Secant</td>
<td>18.46</td>
<td>101</td>
<td>0.75</td>
</tr>
<tr>
<td>trainbfg</td>
<td>BFGS quasi-Newton</td>
<td>10.86</td>
<td>44</td>
<td>1.02</td>
</tr>
<tr>
<td>trainlm</td>
<td>Levenberg-Marquardt</td>
<td>1.87</td>
<td>6</td>
<td>0.46</td>
</tr>
</tbody>
</table>
Improving Generalization

One of the problems that occurs during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations.

The following figure shows the response of a 1-20-1 neural network which has been trained to approximate a noisy sine function. The underlying sine function is shown by the dotted line, the noisy measurements are given by the ‘+’ symbols, and the neural network response is given by the solid line. Clearly this network has overfit the data and will not generalize well.

One method for improving network generalization is to use a network which is just large enough to provide an adequate fit. The larger a network you use, the more complex the functions that the network can create. If we use a small enough network, it will not have enough power to overfit the data. Run the Neural Network Design Demonstration [HDB96] to investigate how reducing the size of a network can prevent overfitting.

The problem is that it is difficult to know beforehand how large a network should be for a specific application. There are two other methods for improving
generalization which are implemented in the Neural Network Toolbox: regularization and early stopping. The next few subsections will describe these two techniques, and the routines to implement them.

**Regularization**
The first method for improving generalization is called regularization. This involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set. The next subsection will explain how the performance function can be modified, and the following subsection will describe two routines which will automatically set the optimal performance function to achieve the best generalization.

**Modified Performance Function**
The typical performance function that is used for training feedforward neural networks is the mean sum of squares of the network errors:

\[
F = \text{mse} = \frac{1}{N} \sum_{i=1}^{N} (e_i)^2 = \frac{1}{N} \sum_{i=1}^{N} (t_i - \hat{a}_i)^2.
\]

It is possible to improve generalization if we modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights and biases:

\[
\text{msereg} = \gamma \text{mse} + (1 - \gamma) \text{msw},
\]

where \( \gamma \) is the performance ratio, and

\[
\text{msw} = \frac{1}{n} \sum_{j=1}^{n} w_j^2.
\]

Using this performance function will cause the network to have smaller weights and biases, and this will force the network response to be smoother and less likely to overfit.

In the following code we re-initialize our previous network and retrain it using the BFGS algorithm with the regularized performance function. Here we set
the performance ratio to 0.5, which gives equal weight to the mean square errors and the mean square weights.

```matlab
net = newff([-1 2; 0 5], [3, 1], {'tansig', 'purelin'}, 'trainbfg');
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 1 1 1];
net = train(net, p, t);
```

The problem with regularization is that it is difficult to determine the optimum value for the performance ratio parameter. If we make this parameter too large, we may get overfitting. If the ratio is too small, the network will not adequately fit the training data. In the next section we will describe two routines which automatically set the regularization parameters.

**Automated Regularization (TRAINBR)**

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [MacK92]. In this framework the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. We can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this users guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [FoHa97].
Bayesian regularization has been implemented in the function `trainbr`. The following code shows how we can train a 1-20-1 network using this function to approximate the noisy sine wave shown earlier in this section.

```matlab
net = newff([-1 1],[20,1],{'tansig','purelin'},'trainbr');
net.trainParam.show = 10;
net.trainParam.epochs = 50;
rnd('seed',192736547);
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
net = init(net);
net = train(net,p,t);
TRAINBR, Epoch 0/50, SSE 107.962/0, Gradient 9.51e+01/1.00e-10,
#Par 6.10e+01/61
TRAINBR, Epoch 10/50, SSE 20.5/0, Gradient 1.88e-01/1.00e-10,
#Par 1.82e+01/61
TRAINBR, Epoch 20/50, SSE 20.5/0, Gradient 4.07e-02/1.00e-10,
#Par 1.65e+01/61
TRAINBR, Epoch 30/50, SSE 20.5/0, Gradient 5.57e-01/1.00e-10,
#Par 1.55e+01/61
TRAINBR, Epoch 40/50, SSE 20.5/0, Gradient 2.76e-01/1.00e-10,
#Par 1.48e+01/61
TRAINBR, Epoch 50/50, SSE 20.5/0, Gradient 4.05e-01/1.00e-10,
#Par 1.42e+01/61

One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 14 parameters (indicated by #Par in the printout), out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain the same, no matter how large the total number of parameters in the network becomes.

The following figure shows the response of the network. In contrast to the previous figure, in which a 1-20-1 network overfit the data, here we see that the network response is very close to the underlying sine function (dotted line), and, therefore, the network will generalize well to new inputs. We could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.
Another method for improving generalization is called early stopping. In this technique, the available data is divided into three subsets. The first subset is the training set which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error will normally decrease during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set will typically begin to rise. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during the training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this may indicate a poor division of the data set.

Early stopping can be used with any of the training functions which were described earlier in this chapter. You simply need to pass the validation data to the training function. The following sequence of commands demonstrates how to use the early stopping function.
First we will create a simple test problem. For our training set we will generate a noisy sine wave with input points ranging from -1 to 1 at steps of 0.05.

\[
p = [-1:0.05:1];
\]
\[
t = \sin(2\pi p) + 0.1 \cdot \text{randn(size(p))};
\]

Next we will generate the validation set. The inputs will range from -1 to 1, as in the test set, but we will offset them slightly. To make the problem more realistic, we also add a different noise sequence to the underlying sine wave. Notice that the validation set is contained in a structure which contains both the inputs and the targets.

\[
v.P = [-0.975:.05:0.975];
\]
\[
v.T = \sin(2\pi v.P) + 0.1 \cdot \text{randn(size(v.P))};
\]

We will now create a 1-20-1 network, as in our previous example with regularization, and train it. (Notice that the validation structure is passed to \texttt{train} after the initial input and layer conditions, which are null vectors in this case since the network contains no delays.) For this example we will use the training function \texttt{traingdx}, although early stopping can be used with any of the other training functions we have discussed in this chapter.

\[
\text{net} = \text{newff([-1 1],[20,1],\{'tansig','purelin'\},\text{\texttt{traingdx}}]);}
\]
\[
\text{net.trainParam.show} = 25;
\]
\[
\text{net.trainParam.epochs} = 300;
\]
\[
\text{net} = \text{init(net)};
\]
\[
[\text{net, tr}] = \text{train}\text{(net, p, t, [], [], v)};
\]

\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 0/300, ME\text{SE} 9.39342/0, Grad\text{ient} 17.7789/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 25/300, ME\text{SE} 0.312465/0, Grad\text{ient} 0.873551/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 50/300, ME\text{SE} 0.102526/0, Grad\text{ient} 0.206456/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 75/300, ME\text{SE} 0.0459503/0, Grad\text{ient} 0.0954717/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 100/300, ME\text{SE} 0.015725/0, Grad\text{ient} 0.0299898/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 125/300, ME\text{SE} 0.00628898/0, Grad\text{ient} 0.042467/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{Epoch} 131/300, ME\text{SE} 0.00650734/0, Grad\text{ient} 0.133314/1e-06
\texttt{R}e\texttt{a}n\texttt{g}d\texttt{x}, \text{V}al\text{i}d\text{ation} stop.
In the following figure we have a graph of the network response. We can see that the network did not overfit the data, as in the earlier example, although the response is not extremely smooth, as when using regularization. This is characteristic of early stopping.
Preprocessing and Postprocessing

Neural network training can be made more efficient if certain preprocessing steps are performed on the network inputs and targets. In this section we describe several preprocessing routines which can be used.

Min and Max (PREMNMX, POSTMNMX, TRAMNMX)

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function premnmx can be used to scale inputs and targets so that they fall in the range [-1,1]. The following code illustrates the use of this function.

```matlab
[pn, minp, maxp, tn, mint, maxt] = premnmx(p, t);
net = train(net, pn, tn);
```

The original network inputs and targets are given in the matrices `p` and `t`. The normalized inputs and targets, `pn` and `tn`, that are returned will all fall in the interval [-1,1]. The vectors `minp` and `maxp` contain the minimum and maximum values of the original inputs, and the vectors `mint` and `maxt` contain the minimum and maximum values of the original targets. After the network has been trained, these vectors should be used to transform any future inputs which are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If `premnmx` is used to scale both the inputs and targets, then the output of the network will be trained to produce outputs in the range [-1,1]. If you want to convert these outputs back into the same units which were used for the original targets, then you should use the routine `postmnmx`. In the following code we simulate the network which was trained in the previous code and then convert the network output back into the original units.

```matlab
an = sim(net, pn);
a = postmnmx(an, mint, maxt);
```

The network output `an` will correspond to the normalized targets `tn`. The un-normalized network output `a` is in the same units as the original targets `t`.

If `premnmx` is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the minimum and maximums which were computed for the training set. This can
be accomplished with the routine `transmx`. In the following code we apply a new set of inputs to the network we have already trained.

```matlab
pnewn = transmx(pnew, minp, maxp);
anewn = sim(net, pnewn);
anew = postmnmx(anewn, mint, maxt);
```

### Mean and Stand. Dev. (PRESTD, POSTSTD, TRASTD)

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. This procedure is implemented in the function `prestd`. It normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of `prestd`.

```matlab
[pn, meanp, stdp, tn, meant, stdt] = prestd(p, t);
```

The original network inputs and targets are given in the matrices `p` and `t`. The normalized inputs and targets, `pn` and `tn`, that are returned will have zero means and unity standard deviation. The vectors `meanp` and `stdp` contain the mean and standard deviations of the original inputs, and the vectors `meant` and `stdt` contain the means and standard deviations of the original targets. After the network has been trained, these vectors should be used to transform any future inputs which are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If `prestd` is used to scale both the inputs and targets, then the output of the network will be trained to produce outputs with zero mean and unity standard deviation. If you want to convert these outputs back into the same units which were used for the original targets, then you should use the routine `poststd`. In the following code we simulate the network which was trained in the previous code and then convert the network output back into the original units.

```matlab
an = sim(net, pn);
a = poststd(an, meant, stdt);
```

The network output `an` will correspond to the normalized targets `tn`. The un-normalized network output `a` is in the same units as the original targets `t`.

If `prestd` is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the means and standard deviations which were computed for the training set. This can be
accomplished with the routine `trastd`. In the following code we apply a new set of inputs to the network we have already trained.

```matlab
pnewn = trastd(pnew, meanp, stdp);
anewn = sim(net, pnewn);
anew = poststd(anewn, meant, stdt);
```

**Principal Component Analysis (PREPCA, TRAPCA)**

In some situations the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis. This technique has three effects: it orthogonalizes the components of the input vectors (so that they are uncorrelated with each other); it orders the resulting orthogonal components (principal components) so that those with the largest variation come first; and it eliminates those components which contribute the least to the variation in the data set. The following code illustrates the use of `prepca`, which performs a principal component analysis.

```matlab
[pn, meanp, stdp] = prestd(p);
[ptrans, transMat] = prepca(pn, 0.02);
```

Note that we first normalize the input vectors, using `prestd`, so that they have zero mean and unity variance. This is a standard procedure when using principal components. In this example, the second argument passed to `prepca` is 0.02. This means that `prepca` will eliminate those principal components which contribute less than 2% to the total variation in the data set. The matrix `ptrans` contains the transformed input vectors. The matrix `transMat` contains the principal component transformation matrix. After the network has been trained, this matrix should be used to transform any future inputs which are applied to the network. It effectively becomes a part of the network, just like the network weights and biases. If you multiply the normalized input vectors `pn` by the transformation matrix `transMat`, you will obtain the transformed input vectors `ptrans`.

If `prepca` is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the transformation matrix which was computed for the training set.
This can be accomplished with the routine \texttt{trapca}. In the following code we apply a new set of inputs to a network we have already trained.

\begin{verbatim}
    pnewn = trastd(pnew, meanp, stdp);
pnewtrans = trapca(pnewn, transMat);
a = sim(net, pnewtrans);
\end{verbatim}

**Post-training Analysis (POSTREG)**

The performance of a trained network can be measured to some extent by the errors on the training, validation and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine \texttt{postreg} is designed to perform this analysis.

The following commands illustrate how we can perform a regression analysis on the network which we previously trained in the early stopping section.

\begin{verbatim}
    a = sim(net, p);
    [m, b, r] = postreg(a, t)
\end{verbatim}

\begin{verbatim}
    m = 0.9874
    b = -0.0067
    r = 0.9935
\end{verbatim}

Here we pass the network output and the corresponding targets to \texttt{postreg}. It returns three parameters. The first two, \( m \) and \( b \), correspond to the slope and the y-intercept of the best linear regression relating targets to network outputs. If we had a perfect fit (outputs exactly equal to targets), the slope would be 1, and the y-intercept would be 0. In this example we can see that the numbers are very close. The third variable returned by \texttt{postreg} is the correlation coefficient (R-value) between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In our example here the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by \texttt{postreg}. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is
indicated by the solid line. In this example it is difficult to distinguish the best linear fit line from the perfect fit line, because the fit is so good.
Sample Training Session

We have covered a number of different concepts in this chapter. At this point it might be useful to put some of these ideas together with an example of how a typical training session might go.

For this example we are going to use data from a medical application [PuLu92]. We want to design an instrument which can determine serum cholesterol levels from measurements of spectral content of a blood sample. We have a total of 264 patients for which we have measurements of 21 wavelengths of the spectrum. For the same patients we also have measurements of hdl, ldl and vldl cholesterol levels, based on serum separation. The first step is to load the data into the workspace and perform a principal component analysis.

```matlab
load choles_all
[pn,meanp,stdp,t,meant,stad] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.001);
```

Here we have conservatively retained those principal components which account for 99.9% of the variation in the data set. Let's check the size of the transformed data.

```matlab
[R,Q] = size(ptrans)
R =
4
Q =
264
```

There was apparently significant redundancy in the data set, since the principal component analysis has reduced the size of the input vectors from 21 to 4.

The next step is to divide the data up into training, validation and test subsets. We will take one fourth of the data for the validation set, one fourth for the test set and one half for the training set. We will pick the sets as equally spaced points throughout the original data.

```matlab
iitst = 2:4:Q;
iival = 4:4:Q;
iitr = [1:4:Q 3:4:Q];
v.P = ptrans(:,iival); v.T = t(:,iival);
t.P = ptrans(:,iitst); t.V = t(:,iitst);
ptr = ptrans(:,iitr); ttr = t(:,iitr);
```
We are now ready to create a network and train it. For this example we will try a two-layer network, with tan-sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This is a useful structure for function approximation (or regression) problems. As an initial guess, we will use five neurons in the hidden layer. The network should have three output neurons since there are three targets. We will use the Levenberg-Marquardt algorithm for training.

```matlab
net = newff(minmax(ptr),[5 3],{'tansig' 'purelin'},'trainlm');
[net,tr]=train(net,ptr,ttr,[],[],v,t);
```

TRAINLM, Epoch 0/100, MSE 3.11023/0, Gradient 804.959/1e-10
TRAINLM, Epoch 15/100, MSE 0.330295/0, Gradient 104.219/1e-10
TRAINLM, Validation stop.

The training stopped after 15 iterations because the validation error increased. It is a useful diagnostic tool to plot the training, validation and test errors to check the progress of training. We can do that with the following commands.

```matlab
plot(tr.epoch,tr.perf,tr.epoch,tr.vperf,tr.epoch,tr.tperf)
legend('Training','Validation','Test',-1);
ylabel('Squared Error'); xlabel('Epoch')
```

The result is shown in the following figure. The result here is reasonable, since the test set error and the validation set error have similar characteristics, and it doesn't appear that any significant overfitting has occurred.
The next step is to perform some analysis of the network response. We will put the entire data set through the network (training, validation and test) and will perform a linear regression between the network outputs and the corresponding targets. First we will need to un-normalize the network outputs.

```
an = sim(net,ptran);
a = poststd(an,meant,stdt);
for i=1:3
    figure(i)
    [m(i),b(i),r(i)] = postreg(a(:,i),t(:,i));
end
```

In this case we have three outputs, so we perform three regressions. The results are shown in the following figures.
Backpropagation
The first two outputs seem to track the targets reasonably well (this is a difficult problem), and the R-values are almost 0.9. The third output (vldl levels) is not well modeled. We probably need to work more on that problem. We might go on to try other network architectures (more hidden layer neurons), or to try Bayesian regularization instead of early stopping for our training technique. Of course there is also the possibility that vldl levels cannot be accurately computed based on the given spectral components.
Limitations and Cautions

The gradient descent algorithm is generally very slow, because it requires small learning rates for stable learning. The momentum variation is usually faster than simple gradient descent, since it allows higher learning rates while maintaining stability, but it is still too slow for many practical applications. These two methods would normally be used only when incremental training is desired. You would normally use Levenberg-Marquardt training if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available.

Multi-layered networks are capable of performing just about any linear or nonlinear computation, and can approximate any reasonable function arbitrarily well. Such networks overcome the problems associated with the perceptron and linear networks. However, while the network being trained may be capable theoretically of performing correctly, backpropagation and its variations may not always find a solution. See page 12-8 of [HDB96] for a discussion of convergence to local minimum points.

Picking the learning rate for a nonlinear network is a challenge. As with linear networks, a learning rate that is too large leads to unstable learning. Conversely, a learning rate that is too small results in incredibly long training times. Unlike linear networks, there is no easy way of picking a good learning rate for nonlinear multilayer networks. See page 12-8 of [HDB96] for examples of choosing the learning rate. With the faster training algorithms, the default parameter values will normally perform adequately.

The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity see the figures on pages 12-5 to 12-7 of [HDB96], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface it is possible for the network solution to become trapped in one of these local minima. This may happen depending on the initial starting conditions. Settling in a local minimum may be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation will not always find the correct weights for the optimum solution. You may wish to re-initialize the network and retrain several times to guarantee that you have the best solution.
Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fit, but the fitting curve takes wild oscillations between these points. Ways of dealing with various of these issues have been discussed in the section on improving generalization. This topic is also discussed starting on page 11-21 of [HDB96].
Summary

Backpropagation can train multilayer feed-forward networks with differentiable transfer functions to perform function approximation, pattern association, and pattern classification. (Other types of networks can be trained as well, although the multilayer network is most commonly used.) The term backpropagation refers to the process by which derivatives of network error, with respect to network weights and biases, can be computed. This process can be used with a number of different optimization strategies.

The architecture of a multilayer network is not completely constrained by the problem to be solved. The number of inputs to the network is constrained by the problem, and the number of neurons in the output layer is constrained by the number of outputs required by the problem. However, the number of layers between network inputs and the output layer and the sizes of the layers are up to the designer.

The two-layer sigmoid/linear network can represent any functional relationship between inputs and outputs if the sigmoid layer has enough neurons.

There are several different backpropagation training algorithms. They have a variety of different computation and storage requirements, and no one algorithm is best suited to all locations. The following list summarizes the training algorithms included in the toolbox.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traingd</td>
<td>Basic gradient descent. Slow response, can be used in incremental mode training.</td>
</tr>
<tr>
<td>traingdm</td>
<td>Gradient descent with momentum. Generally faster than traingd. Can be used in incremental mode training.</td>
</tr>
<tr>
<td>traingdx</td>
<td>Adaptive learning rate. Faster training than traingd, but can only be used in batch mode training.</td>
</tr>
<tr>
<td>trainrp</td>
<td>Resilient backpropagation. Simple batch mode training algorithm with fast convergence and minimal storage requirements.</td>
</tr>
</tbody>
</table>
One problem which can occur when training neural networks is that the network can overfit on the training set and not generalize well to new data outside the training set. This can be prevented by training with \texttt{trainbr}, but it can also be prevented by using early stopping with any of the other training routines. This requires that the user pass a validation set to the training algorithm, in addition to the standard training set.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{traincfg}</td>
<td>Fletcher-Reeves conjugate gradient algorithm. Has smallest storage requirements of the conjugate gradient algorithms.</td>
</tr>
<tr>
<td>\texttt{traincgp}</td>
<td>Polak-Ribière conjugate gradient algorithm. Slightly larger storage requirements than \texttt{traincfg}. Faster convergence on some problems.</td>
</tr>
<tr>
<td>\texttt{traincgb}</td>
<td>Powell-Beale conjugate gradient algorithm. Slightly larger storage requirements than \texttt{traincgp}. Generally faster convergence.</td>
</tr>
<tr>
<td>\texttt{trainscg}</td>
<td>Scaled conjugate gradient algorithm. The only conjugate gradient algorithm that requires no line search.</td>
</tr>
<tr>
<td>\texttt{trainbfg}</td>
<td>BFGS quasi-Newton method. Requires storage of approximate Hessian matrix and has more computation in each iteration than conjugate gradient algorithms, but usually converges in fewer iterations.</td>
</tr>
<tr>
<td>\texttt{trainoss}</td>
<td>One step secant method. Compromise between conjugate gradient methods and quasi-Newton methods.</td>
</tr>
<tr>
<td>\texttt{trainlm}</td>
<td>Levenberg-Marquardt algorithm. Fastest training algorithm for networks of moderate size. Has memory reduction feature for use when the training set is large.</td>
</tr>
<tr>
<td>\texttt{trainbr}</td>
<td>Bayesian regularization. Modification of the Levenberg-Marquardt training algorithm to produce networks which generalize well. Reduces the difficulty of determining the optimum network architecture.</td>
</tr>
</tbody>
</table>
In order to produce the most efficient training, it is often helpful to preprocess the data before training. It is also helpful to analyze the network response after training is complete. The toolbox contains a number of routines for pre- and post-processing. They are summarized in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>premnmx</code></td>
<td>Normalize data to fall in the range [-1,1].</td>
</tr>
<tr>
<td><code>postmnmx</code></td>
<td>Inverse of <code>premnmx</code>. Used to convert data back to standard units.</td>
</tr>
<tr>
<td><code>tramnmx</code></td>
<td>Normalize data using previously computed minimums and maximums. Used to preprocess new inputs to networks which have been trained with data normalized with <code>premnmx</code>.</td>
</tr>
<tr>
<td><code>prestd</code></td>
<td>Normalize data to have zero mean and unity standard deviation.</td>
</tr>
<tr>
<td><code>poststd</code></td>
<td>Inverse of <code>prestd</code>. Used to convert data back to standard units.</td>
</tr>
<tr>
<td><code>trastd</code></td>
<td>Normalize data using previously computed means and standard deviations. Used to preprocess new inputs to networks which have been trained with data normalized with <code>prestd</code>.</td>
</tr>
<tr>
<td><code>prepca</code></td>
<td>Principal component analysis. Reduces dimension of input vector and un-correlates components of input vectors.</td>
</tr>
<tr>
<td><code>trapca</code></td>
<td>Preprocess data using previously computed principal component transformation matrix. Used to preprocess new inputs to networks which have been trained with data transformed with <code>prepca</code>.</td>
</tr>
<tr>
<td><code>postreg</code></td>
<td>Linear regression between network outputs and targets. Used to determine the adequacy of network fit.</td>
</tr>
</tbody>
</table>