22 Practical Training Issues

Objectives

Previous chapters have focused on particular neural network architectures and training rules, with an emphasis on fundamental understanding. In this chapter, we will discuss some practical training tips that apply to a variety of networks. No derivations are provided for the techniques that are presented here, but we have found these methods to be useful in practice.

There will be three basic sections in this chapter. The first section describes things that need to be done prior to training a network, such as collecting and preprocessing data and selecting the network architecture. The second section addresses network training itself. The final section considers post-training analysis.

Theory and Examples

In previous chapters, we have discussed a variety of neural network architectures and learning rules. Those chapters have placed special emphasis on the fundamental concepts behind each network. In this chapter, we will concentrate on practical aspects of training neural networks. Theoretical aspects and practical aspects are not mutually exclusive. It is only by combining a deep knowledge of network fundamentals with practical experience in using neural networks that we can get the most out of this technology.

Figure 22.1 illustrates the neural network training process. It is an iterative procedure that begins by collecting data and preprocessing it to make training more efficient. At this stage, the data also needs to be divided into training/validation/testing sets (see Chapter 13). After the data is selected, we need to choose the appropriate network type (multilayer, competitive, dynamic, etc.) and architecture (e.g., number of layers, number of neurons). Then we select a training algorithm that is appropriate for the network and the problem we are trying to solve. After the network is trained, we want to analyze the performance of the network. This analysis may lead us to discover problems with the data, the network architecture, or the training algorithm. The entire process is then iterated until the network performance is satisfactory.

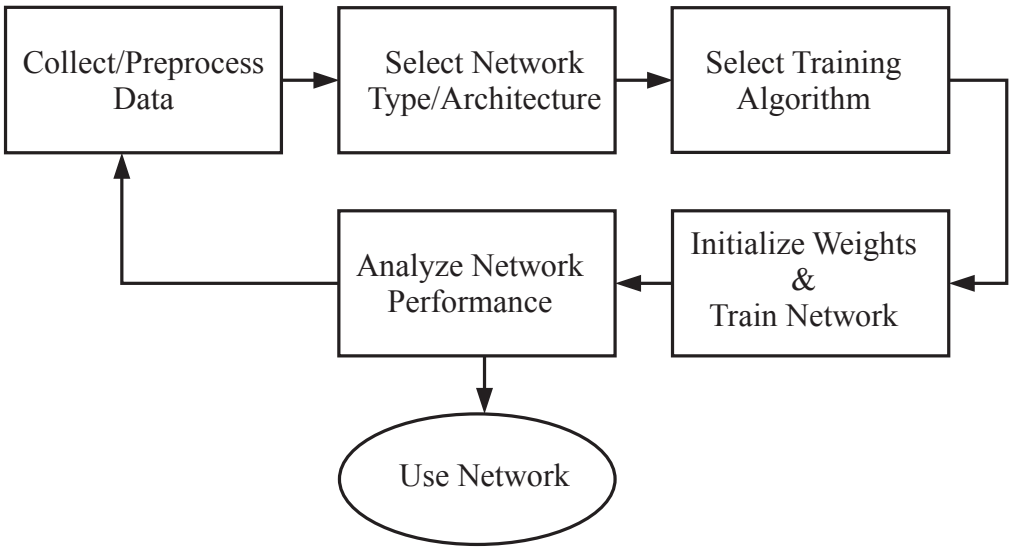

Figure 22.1 Flowchart of Neural Network Training Process

In the remainder of this chapter, we will discuss each part of the training process in some detail. We have divided this material into three major sections: Pre-Training Steps, Training the Network, and Post-Training Analysis.

Before we dig into the details of training, it is worth making a preliminary comment. Before beginning the neural network training process, you should first determine if a neural network is needed to solve your problem, or if some simpler linear technique might be adequate. For example, there is no need to use a neural network for a fitting problem, if standard linear regression will produce a satisfactory result. The neural network techniques provide additional power, but at the expense of more challenging training requirements. When linear methods will work, they are the first choice.

Pre-Training Steps

There are a number of steps that need to be performed before the network is trained. They can be grouped into three categories: Selection of Data, Data Preprocessing, and Choice of Network Type and Architecture.

Selection of Data

It is generally difficult to incorporate prior knowledge into a neural network, therefore the network will only be as good as the data that is used to train it. Neural networks represent a technology that is at the mercy of the data. The training data must span the full range of the input space for which the network will be used. As we discussed in Chapter 13, there are training methods we can use to insure that the network interpolates accurately throughout the range of the data provided (generalizes well). However, it is not possible to guarantee network performance when the inputs to the network are outside the range of the training set. Neural networks, like other nonlinear “black box” methods, do not extrapolate well.

It is not always easy to be sure that the input space is adequately sampled by the training data. For simple problems, in which the dimension of the input vector is small, and each element of the input vector can be chosen independently, we can sample the input space using a grid. However, these conditions are not often satisfied. For many problems, the dimension of the input space is large, which precludes the use of grid sampling. In addition, it is often the case that the input variables are dependent. For example, consider Figure 22.2. The shaded area represents the range over which the two inputs can vary. Even though each variable can range from -1 to 1, we would not need to create a grid in which both variables vary throughout their range, as shown by the dots in Figure 22.2. The network only needs to fit the function in the shaded area, since this is where the network will be used. It would be inefficient to fit the network outside the range of its use. This is especially true when the input dimension is large.

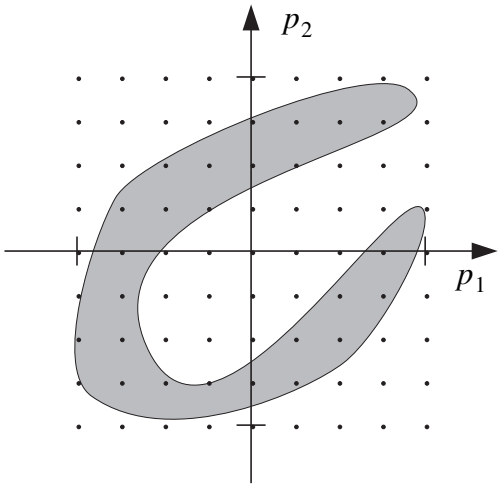

Figure 22.2 Input Range With Dependent Input Variables

It may not be possible to precisely define the active region of the input space. However, we can often collect data during standard operation of the system we are trying to model. In some cases, we have complete control over the design of the experiment during which data is collected. In these cases, we must be sure that the experimental setup drives the system through all conditions for which we plan to use the network.

How can we be sure that the input space has been adequately sampled by the training data? This is difficult to do prior to training, and there are many cases in which we have no control over the data collection process and must use whatever data is available. We will come back to this subject in the Post-Training Analysis section on page 22-18. By analyzing the trained network, we can often tell if the training data was sufficient. In addition, we can use techniques that indicate when a network is being used outside the range of the data with which it was trained. This will not improve the network performance, but it will prevent us from using a network in situations where it is not reliable.

After collecting the data, we generally divide it into three sets: training, validation and testing. As we discussed in Chapter 13, the training set will generally make up approximately 70% of the full data set, with validation and testing making up approximately 15% each. It is important that each of these sets be representative of the full data set — that the validation and test sets cover the same region of the input space as the training set. The simplest method for dividing the data is to select each set at random from the full data set. This usually produces a good result, but it is best to review the division to check for major differences between the sets. It is also possible in the post-training analysis to detect problems in the data division. We will have more to say about that later.

A final question we must ask about the selection of data is “Do we have enough data?” This is difficult to answer, especially before we train the net-

work. The amount of data that is required depends on the complexity of the underlying function that we are trying to approximate (or the complexity of the decision boundaries that we are trying to implement). If the function to be approximated is very complex, with many inflection points, then this requires a large amount of data. If the function is very smooth, then the data requirements are significantly reduced (unless the data is very noisy). The choice of the data set size is closely related to the choice of the number of neurons in the neural network. This is discussed in the Choice of Network Architecture section on page 22-8. Of course, we generally don't know how complex the underlying function is before we begin training the network. For this reason, as we discussed earlier, the entire neural network training process is iterative. At the completion of training we will analyze the performance of the network. The results of that analysis can help us decide if we have enough data or not.

Data Preprocessing

The main purpose of the data preprocessing stage is to facilitate network training. Data preprocessing consists of such steps as normalization, nonlinear transformations, feature extraction, coding of discrete inputs/targets, handling of missing data, etc. The idea is to perform preliminary processing of the data to make it easier for the neural network training to extract the relevant information.

For example, in multilayer networks, sigmoid transfer functions are often used in the hidden layers. These functions become essentially saturated when the net input is greater than three ($\exp(-3) \cong 0.05$). We don't want this to happen at the beginning of the training process, because the gradient will then be very small. In the first layer, the net input is a product of the input times the weight plus the bias. If the input is very large, then the weight must be small in order to prevent the transfer function from becoming saturated. It is standard practice to normalize the inputs before applying them to the network. In this way, initializing the network weights to small random values guarantees that the weight-input product will be small. Also, when the input values are normalized, the magnitudes of the weights have a consistent meaning. This is especially important when using regularization, as described in Chapter 13. Regularization requires the weight values to be small. However, "small" is a relative term; if the input values are very small, we need large weights to produce a significant net input. Normalizing the inputs clarifies the meaning of "small" weights.

Normalization

There are two standard methods for *normalization*. The first method normalizes the data so that they fall into a standard range — typically -1 to 1. This can be done with

$$\mathbf{p}^n = 2(\mathbf{p} - \mathbf{p}^{min})./(\mathbf{p}^{max} - \mathbf{p}^{min}) - 1, \tag{22.1}$$

where $\mathbf{p}^{min}$ is the vector containing the minimum values of each element of the input vectors in the data set, $\mathbf{p}^{max}$ contains the maximum values, ./

represents an element-by-element division of two vectors, and $\mathbf{p}^n$ is the resulting normalized input vector.

An alternative normalization procedure is to adjust the data so that they have a specified mean and variance — typically 0 and 1. This can be done with the transformation

$$\mathbf{p}^n = (\mathbf{p} - \mathbf{p}^{mean})./\mathbf{p}^{std}, \tag{22.2}$$

where $\mathbf{p}^{mean}$ is the average of the input vectors in the data set, and $\mathbf{p}^{std}$ is the vector containing the standard deviations of each element of the input vectors.

Generally, the normalization step is applied to both the input vectors and the target vectors in the data set.

Nonlinear Transformation

In addition to normalization, which involves a linear transformation, *nonlinear transformations* are sometimes also performed as part of the preprocessing stage. Unlike normalization, which is a standard process that can be applied to any data set, these nonlinear transformations are case-specific. For example, many economic variables show a logarithmic dependence [BoJe94]. In that case, it might be appropriate to take the logarithm of the input values before applying them to the neural network. Another example is molecular dynamics simulation [RaMa05], in which atomic forces are calculated as functions of distances between atoms. Since it is known that the forces are inversely related to the distances, we might perform the reciprocal transformation on the inputs, before applying them to the network. This represents one way of incorporating prior knowledge into neural network training. If the nonlinear transformation is cleverly chosen, it can make the network training more efficient. The preprocessing will off-load some of the work required of the neural network in finding the underlying transformation between inputs and targets.

Feature Extraction

Another data preprocessing step is called *feature extraction*. This generally applies to situations in which the dimension of the raw input vectors is very large and the components of the input vector are redundant. The idea of feature extraction is to reduce the dimension of the input space by calculating a small set of features from each input vector, and using the features as the input to the neural network. For example, neural networks can be used to analyze EKG (electrocardiogram) signals to identify heart problems [HeOh97]. The EKG might involve 12 or 15 signals (leads) measured over several minutes at a high sampling rate. This is too much data to apply directly to the neural network. Instead, we would extract certain features from the EKG signal, such as average time intervals between certain waveforms, average amplitudes of certain waves, etc. (See Chapter 25.)

Principal Components

There are also certain general-purpose feature extraction methods. One of these is the method of *principal component analysis* (PCA) [Joll02]. This method transforms the original input vectors so that the components of the

transformed vectors are uncorrelated. In addition, the components of the transformed vector are ordered such that the first component has the greatest variance, the second component has the next greatest variance, etc. We generally keep only the first few components of the transformed vector, which account for most of the variance in the original vector. This results in a large reduction in the dimension of the input vector, if the original components are highly correlated. The drawback of using PCA is that it only considers linear relationships between the components of the input vector. When reducing the dimension using a linear transformation, we might lose some nonlinear information. Since the main purpose of using neural networks is to gain the power of their nonlinear mapping capabilities, we should be careful when using principal components to reduce the input dimension before applying the inputs to the neural network. There is a nonlinear version of PCA, called kernel PCA [ScSm99].

Coding the Targets

Another important preprocessing step is needed whenever the inputs or targets take on only discrete values. For example, in pattern recognition problems, each target will represent one of a finite number of classes. In these cases we need to have a procedure for *coding the inputs or targets*. If we have a pattern recognition problem in which there are four classes, there are at least three common ways in which we could code the targets. First, we can have scalar targets that take on four possible values (e.g., 1, 2, 3, 4). Second, we can have two-dimensional targets, which represent a binary code of the four classes (e.g., (0,0), (0,1), (1,0), (1,1)). Third, we can have four-dimensional targets, in which only one neuron at a time is active (e.g., (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)). The third method tends to yield the best results in our experience. (Note that discrete inputs can be coded in the same ways as discrete targets.)

When coding the target values, we also need to consider the transfer function that is used in the output layer of the network. For pattern recognition problems, we would typically use sigmoid functions: log-sigmoid or tangent-sigmoid. If we use the tangent-sigmoid in the last layer, which is more common, then we might consider assigning target values to -1 or 1, which represent the asymptotes of the function. However, this tends to cause difficulties for the training algorithm, which tries to saturate the sigmoid function to meet the target value. It is better to assign target values at the point where the second derivative of the sigmoid function is maximum (see [LeCu98]). For the tangent-sigmoid function, this occurs when the net input is -1 and 1, which corresponds to output values of -0.76 and +0.76.

Softmax

Another transfer function that is used in the output layer of multilayer pattern recognition networks is the *softmax* function. This transfer function has the form

$$a_i = f(n_i) = \exp(n_i) \div \sum_{j=1}^{S} \exp(n_j). \qquad (22.3)$$

The outputs of the softmax transfer function can be interpreted as the probabilities associated with each class. Each output will fall between 0 and 1, and the sum of the outputs will equal 1. See Chapter 24 for an example application of the softmax transfer function.

Missing Data

Another practical issue to consider is *missing data*. It is often the case, especially when dealing with economic data, for example, that some of the data is missing. For instance, we might have an input vector containing 20 economic variables that are collected at monthly intervals. There may be some months in which one or two of the 20 variables were not collected properly. The simplest solution to this problem would be to throw out the data for any month in which any of the variables were missing. However, we might be very limited in the amount of data available, and it might be very expensive to collect additional data. In these cases, we would like to make full use of any data that we have, even if it is incomplete.

There are several strategies for dealing with missing data. If the missing data occurs in an input variable, one possibility is to replace the missing value with the average value for that particular input variable. At the same time, we could add an additional flag element to the input vector that would indicate that missing data for that input variable had been replaced with the average. This additional element of the input vector could be assigned the value 1 when the input variable was available, and 0 when the input variable was missing for that case. This would provide the neural network with information about which variables were missing. An additional flag element would be added to the input vector for every input variable that contained missing points.

If the missing data occurs in an element of the target, then the performance index can be modified so that errors associated with the missing target values are not included. All known target values will contribute to the performance index, but missing target values will not.

Choice of Network Architecture

The next step in the network training process is the choice of network architecture. The basic type of network architecture is determined by the type of problem we wish to solve. Once the basic architecture is chosen, we need to decide such specific details as how many neurons and layers we want to use, how many outputs the network should have, and what type of performance function we want to use for training.

Choice of Basic Architecture

The first step in choosing the architecture is to define the problem that we are trying to solve. For this chapter, we will limit our discussion to four types of problems: fitting, pattern recognition, clustering and prediction.

Fitting

Fitting is also referred to as function approximation or regression. In fitting problems, you want a neural network to map between a set of inputs and a

corresponding set of targets. For example, a realtor might want to estimate home prices from such input variables as tax rate, pupil/teacher ratio in local schools and crime rate. An automotive engineer might want to estimate engine emission levels based on measurements of fuel consumption and speed. A physician might want to predict a patient's body fat level based on body measurements. For fitting problems, the target variable takes on continuous values. (For an example of training a neural network for a fitting problem, see Chapter 23.)

The standard neural network architecture for fitting problems is the multilayer perceptron, with tansig neurons in the hidden layers, and linear neurons in the output layer. The tansig transfer function is generally preferred over the logsig transfer function in the hidden layers for the same reason that inputs are normalized. It produces outputs (which are inputs to the next layer) that are centered near zero, whereas the logsig transfer function always produces positive outputs. For most fitting problems, a single hidden layer is sufficient. If the results with one hidden layer are not satisfactory, two layers are sometimes used. It would be rare in a standard fitting problem to use more than two hidden layers, although, for very difficult problems, deep networks, with many layers, have been used. Linear transfer functions are used in the output layer for fitting problems, because the target output is a continuous variable. As we saw in Chapter 11, a two layer network with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer is a universal approximator.

Radial basis networks can also be used for fitting problems. The Gaussian transfer function is most commonly used in the hidden layer for these networks, with linear transfer functions in the output layer.

Pattern Recognition

Pattern recognition is also referred to as pattern classification. In pattern recognition problems, you want a neural network to classify inputs into a set of target categories. For example, a wine dealer might want to recognize the vineyard that a particular bottle of wine came from, based on a chemical analysis of the wine. A physician might want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness and mitosis.

In addition to fitting problems, multilayer perceptrons can be used for pattern recognition. The main difference between a fitting network and a pattern recognition network is the transfer function used in the output layer. For pattern recognition problems, we generally use a sigmoid function in the output layer. The radial basis function network can also be used for pattern recognition.

For an example of training a neural network for a pattern recognition problem, see Chapter 25.

Clustering

Clustering, or segmentation, is another use for neural networks. In clustering problems, you want a neural network to group data by similarity. For example, businesses may wish to perform market segmentation, which is

done by grouping people according to their buying patterns. Computer scientists may want to perform data mining by partitioning data into related subsets. Biologists may wish to perform bioinformatic analyses, such as grouping genes with related expression patterns.

Any of the competitive networks described in Chapter 16 could be used for clustering. The self-organizing feature map (SOFM) is the most popular network for clustering. The main advantage of the SOFM is that it allows visualization of high-dimensional spaces.

For an example of training a neural network for a clustering problem, see Chapter 26.

Prediction

Prediction also falls under the categories of time series analysis, system identification, filtering or dynamic modeling. The idea is that we wish to predict the future value of some time series. An equities trader might want to predict the future value of some security. A control engineer might want to predict a future value of the concentration of some chemical, which is the output of a processing plant. A power systems engineer might want to predict outages on the electric grid.

Prediction requires the use of dynamic neural networks, as discussed in Chapter 14. The specific form of the network will depend on the particular application. The simplest network for nonlinear prediction is the focused time-delay neural network, which is shown in Figure 22.3. This is part of a general class of dynamic networks, called focused networks, in which the dynamics appear only at the input layer of a static multilayer feedforward network. This network has the advantage that it can be trained using static backpropagation algorithms, since the tapped-delay-line at the input of the network can be replaced with an extended vector of delayed values of the input.

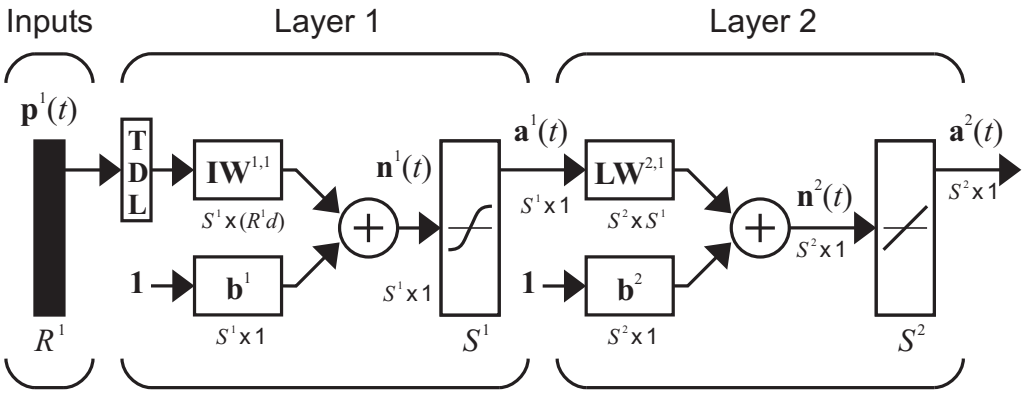

Figure 22.3 Focused Time-Delay Neural Network

For problems of dynamic modeling and control, the NARX network (Nonlinear AutoRegressive model with eXogenous input) is popular. This network is shown in Figure 22.4. The input signal could represent, for

example, the voltage applied to a motor, and the output could represent the angular position of a robot arm. As with the focused time-delay neural network, the NARX network can be trained with static backpropagation. The two tapped-delay-lines can be replaced with extended vectors of delayed inputs and targets. We can use targets, instead of feeding back the network outputs (which would require dynamic backpropagation for training), because the output of the network should match the targets when training is complete.This is discussed in more detail in Chapter 27.

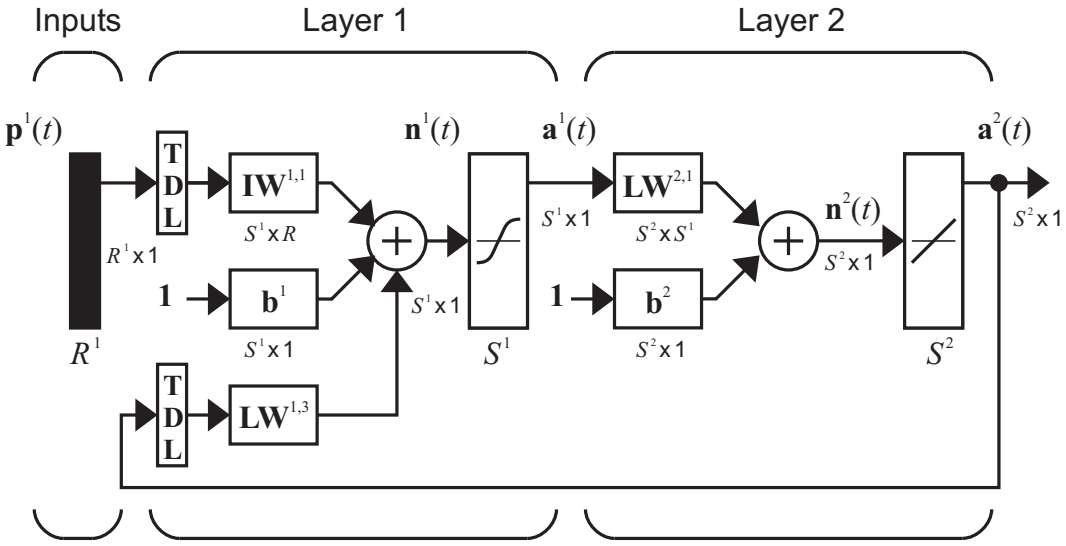

Figure 22.4 NARX Neural Network

There are many other types of dynamic networks that could be used for prediction, but the focused time delay network and the NARX network are the simplest of their type.

For an example of training a neural network for a prediction problem, see Chapter 27.

Selection of Architecture Specifics

After the basic network structure is chosen, we want to select the specifics of the architecture (e.g., the number of layers, the number of neurons, etc.). In some cases, the basic architecture choice will automatically determine the number of layers. For example, if the SOFM is used for clustering, then the network will have one layer. In the case of the multilayer network, which can be used for fitting or pattern recognition, the number of hidden layers is not determined by the problem, since any number of hidden layers is possible. The standard procedure is to begin with a network with one hidden layer. If the performance of the two-layer network is not satisfactory, then a three-layer network can be used. It would be unusual to use more than two hidden layers. The training becomes more difficult when multiple hidden layers are used. This is because each layer performs a squashing operation, as sigmoid functions are used in the hidden layers. This causes the

derivatives of the performance function with respect to weights in the early layers to be quite small, which can cause slow convergence for steepest descent optimization. For very difficult problems, however, deep networks, with several hidden layers, can be used. Typically, parallel or GPU computing is required to train deep multilayer networks within a reasonable amount of time.

We also need to select the number of neurons in each layer. The number of neurons in the output layer is the same as the size of the target vector. The numbers of neurons in the hidden layers are determined by the complexities of the function that is being approximated or the decision boundaries that are being implemented. Unfortunately, we don't normally know how complex the problem is until we try to train the network. The standard procedure is to begin with more neurons than necessary and then to use early stopping or Bayesian regularization to prevent overfitting, as was described in Chapter 13.

The principal drawback to having too many neurons is that the network may overfit the data. If we use early stopping or Bayesian regularization, then we can prevent overfitting. However, there may be some situations in which we are concerned with the computation time or space required by the network (e.g., for real-time implementation on microcontrollers, VLSI or FPGAs). In these cases we want to find the simplest network that will fit the data. If you use Bayesian regularization, the effective number of parameters can be used to determine how many neurons to use. If, after training, the effective number of parameters is much less than the total number of parameters in the network, then the number of neurons can be reduced, and the network retrained. It is also possible to use "pruning" methods to eliminate neurons or weights in the network.

The number of neurons in the last layer is equal to the number of elements in the target vector. However, when there are multiple targets, we have a choice to make. We can have one network with multiple outputs, or we can have multiple networks, each with one output. For example, neural networks have been used to estimate LDL, VLDL and HDL cholesterol levels, based on a spectral analysis of the blood. It is possible to have one neural network with three neurons in the output layer to estimate all three cholesterol levels, or we could have three neural networks, with each one estimating only one of the three components. Theoretically, both methods should work, but in practice one method may work better than another. We generally start with one multi-output network, and then use multiple single-output networks if the original results are not satisfactory.

Another architectural choice is the size of the input vector. This is often a simple choice, which is determined by the training data. However, there are times when input vectors in the training data have redundant or irrelevant elements. When the dimension of the potential input vector is very large, it is sometimes advantageous to eliminate redundant or irrelevant elements. This can reduce the required computation and can assist in pre-

Input Selection

venting overfitting during training. The *input selection* process for nonlinear networks can be quite difficult, and there is no perfect solution. The Bayesian regularization method (Eq. (13.23)) can be modified to assist in input selection. It is possible to have different α parameters for different sets of weights. For example, we can let each column of the weight matrix in the first layer of a multilayer network have its own α parameter. If a given element of the input vector is irrelevant, then the corresponding α parameter would become large and force all elements of that column of the weight matrix to be small. That element could then be eliminated from the input vector.

Another technique that can assist in pruning the input vector is a sensitivity analysis of the trained network. In the Sensitivity Analysis section on page 22-28 we discuss this technique.

Training the Network

After the data has been prepared, and the network architecture has been selected, we are ready to train the network. In this section, we will discuss some of the decisions that need to be made as part of the training process. This includes the method for initializing the weights, the training algorithm, the performance index, and the criterion for stopping training.

Weight Initialization

Before training the network, we need to initialize the weights and biases. The method we use will depend on the type of network. For multilayer networks, the weights and biases are generally set to small random values (e.g., uniformly distributed between -0.5 and 0.5, if the inputs are normalized to fall between -1 and 1). As we discussed in Chapter 12, if we set the weights and biases to zero, the initial condition may fall on a saddle point of the performance surface. If we make the initial weights large, the initial condition can fall on a flat part of the performance surface, caused by saturation of the sigmoid transfer functions.

There is another approach to setting the initial weights and biases for a two-layer network. It was introduced by Widrow and Nguyen [WiNg90]. The idea is to set the magnitude of the weights in the first layer so that the linear region of each sigmoid function covers $1/S^1$ of the range of the input. The biases are then randomly set, so that the center of each sigmoid function falls randomly in the input space. The details of the method are as follows (assuming the inputs to the network have been normalized to values between -1 and 1).

Set row i of $\mathbf{W}^1$, ${}_i\mathbf{w}^1$, to have a random direction and a magnitude of

$$\left\| {}_i\mathbf{w}^1 \right\| = 0.7(S^1)^{1/R}.$$

Set b_i to a uniform random value between $-\left\|{}_i\mathbf{w}^1\right\|$ and $\left\|{}_i\mathbf{w}^1\right\|$.

For competitive networks, the weights can also be set as small random numbers. Another possibility is to randomly select some of the input vectors in the training set to become initial rows of the weight matrix. In this way, we can be sure that the initial weights will fall within the range of the input vectors, so we will be less likely to have dead units, as described in Chapter 16. For the SOM, dead units are not a problem. The initial neighborhood size is set large enough so that all neurons will have the opportunity to learn during the initial stages of training. This will move all weight vectors into the appropriate region of the input space. Training can converge faster, however, if rows of the weight matrix are initially placed in the active input region.

Choice of Training Algorithm

For multilayer networks, we generally use gradient- or Jacobian-based algorithms, as described in Chapter 12. These algorithms can be implemented in either batch mode or sequential (also known as incremental, pattern or stochastic) mode. For example, in the sequential form of steepest descent (see Eq. (11.13)) we update the weights after each input is presented to the network. In batch mode (see page 12-7), all of the inputs are presented to the network, and the total gradient is computed by summing the gradients for each input, before the weights are updated. In some situations, the sequential form is preferred — for example, when on-line or adaptive operation is required. However, many of the more efficient optimization algorithms (e.g., conjugate gradient and Newton's methods) are inherently batch algorithms.

For multilayer networks with up to a few hundred weights and biases that are being used for function approximation, the Levenberg-Marquardt algorithm (see Eq. (12.31)) is usually the fastest training method. When the number of weights reaches a thousand or more, the Levenberg-Marquardt algorithm is not as efficient as some of the conjugate gradient algorithms. This is mainly because the matrix inverse calculation scales geometrically with the number of weights. For large networks, the Scaled Conjugate Gradient algorithm of [Moll93] is very efficient. This method is also attractive for pattern recognition problems. The Levenberg-Marquardt algorithm does not work as well for pattern recognition, in which the sigmoid transfer functions in the final layer are operating well outside the linear region.

Of the algorithms that can be implemented in sequential mode, the fastest are the extended Kalman filter algorithms. These algorithms are closely related to sequential implementations of the Gauss-Newton algorithm. Unlike the batch version of Gauss-Newton, they do not require an inversion of the approximate Hessian matrix. The decoupled extended Kalman filter implementation of [PuFe97] appears to be the most efficient of these types of algorithms.

Stopping Criteria

For most applications of neural networks, the training error never converges identically to zero. The error can reach zero for the perceptron network, assuming a linearly separable problem, as we showed in Chapter 4. However, it is unlikely to happen for multilayer networks. For this reason, we need to have other criteria for deciding when to stop the training.

We can stop the training when the error reaches some specified limit. However, it is usually difficult to know what an acceptable error level is. The simplest criterion is to stop the training after a fixed number of iterations. Because it is also difficult to know how many iterations will be required, the maximum iteration number is generally set reasonably high. If the weights have not converged after the maximum number of iterations has been reached, we can restart training, using the final weights from the first run as initial conditions for the restart. (We will talk more about how to tell if a network has converged in the Post-Training Analysis section on page 22-18.)

Another stopping criterion is the norm of the gradient of the performance index. If this norm reaches a sufficiently small threshold, then the training can be stopped. Since the gradient should be zero at a minimum of the performance index, this criterion will stop the algorithm when it gets close to the minimum. Unfortunately, as we have seen in Chapter 12, the performance surface for multilayer networks can have many flat regions, where the norm of the gradient will be small. For this reason, the threshold for the minimum norm should be set to a very small value (e.g., 10^{-6} for mean square error performance indices, with normalized targets), so that the training does not end prematurely.

We can also stop the training when the reduction in the performance index per iteration becomes small. As with the norm of the gradient, this criterion can stop the training too early. During the training of multilayer networks, the performance can remain almost constant for a number of iterations before dropping suddenly. When training is complete, it is useful to view the training performance curve on a log-log scale, as in Figure 22.5, to verify convergence.

If we are using early stopping, as discussed in Chapter 13, to prevent overfitting, then we will stop the training when the performance on the validation set increases for a set number of iterations. In addition to preventing overfitting, this stopping procedure also provides a significant reduction in computation; for most practical problems, the validation error will increase before any of the other stopping criteria are reached.

As shown in Figure 22.1, neural network training is an iterative process. Even after the training algorithm has converged, post-training analysis may suggest that the network be modified and retrained. In addition, several training runs should be made for each potential network to ensure that a global minimum has been reached.

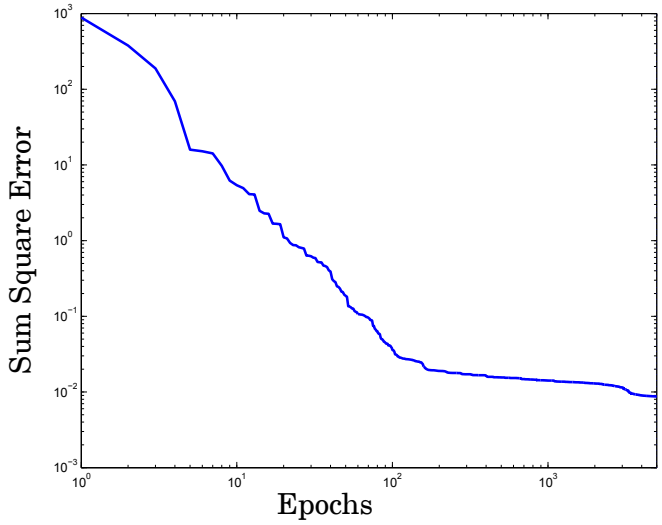

Figure 22.5 Typical Training Performance Curve

The previous stopping criteria apply mainly to gradient-based training. When training competitive networks, like the SOFM, there is no explicit performance index or gradient to monitor for convergence. The training stops only when the maximum number of iterations has been reached. For SOFMs, the learning rate and the neighborhood size are decreased over time. Typically, the neighborhood size is reduced to zero by the completion of training, so the maximum iteration number determines the end of training, as well as the rate of decrease in neighborhood size and learning rate. This is therefore a very important parameter. It is generally chosen to be more than ten times the number of neurons in the network. This is an approximate number, and the network needs to be analyzed at the completion of training to determine if the performance is satisfactory. (This will be discussed in the Post-Training Analysis section on page 22-18.) The network may need to be trained several times with different training lengths to achieve a satisfactory result.

Choice of Performance Function

For multilayer networks, the standard performance index is mean square error. When all inputs in the training set are equally likely to occur, then this can be written

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q), \tag{22.4}$$

or

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}(t_{i,q}-a_{i,q})^2. \tag{22.5}$$

The scale factor that occurs outside the sum has no effect on the location of the optimum weights. Therefore, the sum square error performance index will produce the same weights as the mean square error performance index. However, the appropriate scaling can be useful when comparing errors on data sets of different size.

While mean square error is the most common performance index, there are others that have been used. For example, we could use mean absolute error. This would be similar to Eq. (22.5), except that the absolute value of the error would be used, instead of the square of the error. This performance index is generally less sensitive to one or two large errors in the data set, and is therefore somewhat more robust to outliers than is the mean square error algorithm. This concept can be extended to any power of the absolute error, as follows

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}|t_{i,q}-a_{i,q}|^K, \tag{22.6}$$

where $K = 2$ corresponds to mean square error and $K = 1$ corresponds to mean absolute error. The general error given by Eq. (22.6) is referred to as the Minkowski error.

As we saw in Chapter 13, the mean square performance index can be augmented with the mean square weights, to produce a regularized performance index, which is used to prevent overfitting. The Bayesian regularization algorithm is an excellent training method for preventing overfitting. It uses a regularized performance index, and uses Bayesian methods to select the regularization parameter. See Chapter 13 for details.

Mean square error works well for function approximation problems, in which the target values are continuous. However, in pattern recognition problems, where the targets take on discrete values, other performance indices might be more appropriate. One performance index that has been proposed for classification problems is *cross-entropy* [Bish95]. Cross-entropy is defined as

Cross-Entropy

$$F(\mathbf{x}) = -\sum_{q=1}^{Q}\sum_{i=1}^{S^M} t_{i,q}\ln\frac{a_{i,q}}{t_{i,q}}. \tag{22.7}$$

Here we assume that the target values are 0 and 1, and they identify which of the two classes the input vector belongs to. The softmax transfer function

is generally used in the last layer of the neural network, if the cross-entropy performance index is used.

As a closing note on the choice of performance index, recall from Chapter 11 that the backpropagation algorithm for computing training gradients will work for any differentiable performance index. If you change the performance index, you only need to change the initialization of the sensitivities in the last layer (see Eq. (11.37)).

Multiple Training Runs and Committees of Networks

A single training run may not produce optimal performance, because of the possibility of reaching a local minimum of the performance surface. It is best to restart the training at several different initial conditions and select the network that produces the best performance. Five to ten restarts will almost always produce a global optimum [HaBo07].

There is another way to perform multiple training runs and make use of all of the networks that have been trained. This is called the committee of networks [PeCo93]. For each training session, the validation set is randomly selected from the training data, and a random set of initial weights and biases is chosen. After N networks have been trained, all of the networks are used together to form a joint output. For function approximation networks, the joint output can be a simple average of the outputs of each network. For classification networks, the joint output can be the result of a vote, in which the class that is chosen by the majority of the networks is selected as the output of the committee. The performance of the committee will usually be better than even the best of the individual networks. In addition, the variation in the outputs of the individual networks can be used to provide error bars, or confidence levels, for the committee output.

Post-Training Analysis

Before using a trained neural network, we need to analyze it to determine if the training was successful. There are many techniques for post-training analysis. We will discuss some of the more common ones. Since these techniques vary, depending on the application, we will organize them according to these four application areas: fitting, pattern recognition, clustering and prediction.

Fitting

One useful tool for analyzing neural networks trained for fitting problems is a regression between the trained network outputs and the corresponding targets. We fit a linear function of the form

$$a_q = mt_q + c + \varepsilon_q, \tag{22.8}$$

where m and c are the slope and offset, respectively, of the linear function, t_q is a target value, a_q is a trained network output, and ε_q is the residual error of the regression.

The terms in the regression can be computed as follows:

$$\hat{m} = \frac{\sum_{q=1}^{Q} (t_q - \bar{t})(a_q - \bar{a})}{\sum_{q=1}^{Q} (t_q - \bar{t})^2}, \tag{22.9}$$

$$\hat{c} = \bar{a} - \hat{m}\bar{t}, \tag{22.10}$$

where

$$\bar{a} = \frac{1}{Q}\sum_{q=1}^{Q} a_q, \; \bar{t} = \frac{1}{Q}\sum_{q=1}^{Q} t_q. \tag{22.11}$$

Figure 22.6 shows an example regression analysis. The blue line represents the linear regression, the thin black line represents the perfect match $a_q = t_q$, and the circles represent the data points. In this example, we can see that the match is pretty good, although not perfect. The next step would be to investigate data points that fall far from the regression line. For example, there are two points around $t = 27$ and $a = 17$ that seem to be *outliers*. We would investigate these points to see if there was a problem with the data. This could be a bad data point, or it could be located far from other training points. In the latter case, we would need to collect more data in that region.

Outliers

In addition to computing the regression coefficients, we often also compute the correlation coefficient between the t_q and a_q, which is also known as the R value:

$$R = \frac{\sum_{q=1}^{Q} (t_q - \bar{t})(a_q - \bar{a})}{(Q-1)s_t s_a}, \tag{22.12}$$

where

$$s_t = \sqrt{\frac{1}{Q-1}\sum_{q=1}^{Q} (t_q - \bar{t})} \text{ and } s_a = \sqrt{\frac{1}{Q-1}\sum_{q=1}^{Q} (a_q - \bar{a})}. \tag{22.13}$$

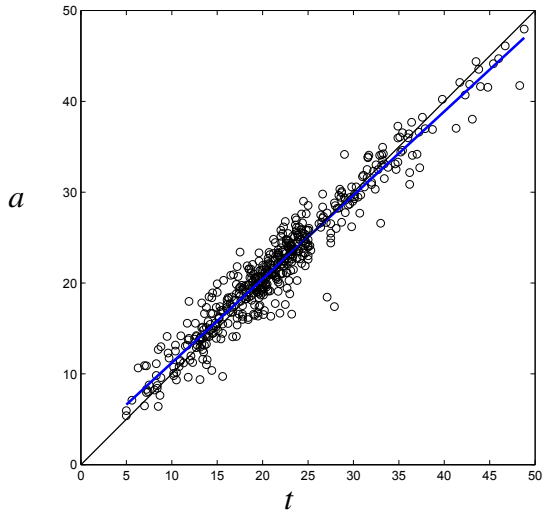

Figure 22.6 Regression Between Trained Network Outputs and Targets

The R value can generally range from -1 to 1, but we would expect it to be close to 1 for our neural network application. If $R = 1$, then all of the data points will fall exactly on the regression line. If $R = 0$, then the data will not be concentrated around the regression line, but will be randomly scattered. For the data of Figure 22.6, $R = 0.965$. We can see that the data does not fall exactly on the regression line, but the variation is relatively small.

The square of the correlation coefficient, R^2, is sometimes used instead of R. R^2 represents the proportion of the variability in a data set that is accounted for by the linear regression, and is also referred to as the coefficient of determination. For the data of Figure 22.6, $R^2 = 0.931$.

When the R or R^2 values are significantly less than 1, then the neural network has not done a good job of fitting the underlying function. A close analysis of the scatter plot may be helpful in determining problems in the fit. For example, we might find that when the targets are large there is more spread in the scatter plot. (This is not the case in Figure 22.6.) We might also notice that there are fewer data points with large targets. This would indicate that we need to have more data points in the training set for these target values.

Recall that the original data set was divided into training, validation (if early stopping is used) and testing subsets. The regression analysis should be performed on each subset individually, as well as the full data set. Differences between the subsets would indicate overfitting or extrapolation. For example, if the training set shows accurate fitting, but the validation and test results are poor, then this would indicate overfitting (which can sometimes happen, even when early-stopping is used). In this case, we might reduce the size of the neural network and retrain. If both the training and validation results are good, but the testing results are poor, then

this could indicate extrapolation (where the testing data falls outside the training and validation data). In this case, we need to provide more data for training and validation. If the results for all three data sets are poor, then it might be necessary to increase the number of neurons in the network. Another choice is to increase the number of layers in the network. If you start with a single hidden layer, and the results are poor, then a second hidden layer could be helpful. First, try more neurons in the single hidden layer, and then increase the number of layers.

In addition to the regression/scatter plot, another tool that can identify outliers is a histogram of the errors, as shown in Figure 22.7. The *y*-axis represents the number of errors that falls within each interval on the *x*-axis. Here we can see that two errors are greater than 8. These represent the same two errors that we identified as outliers in Figure 22.6.

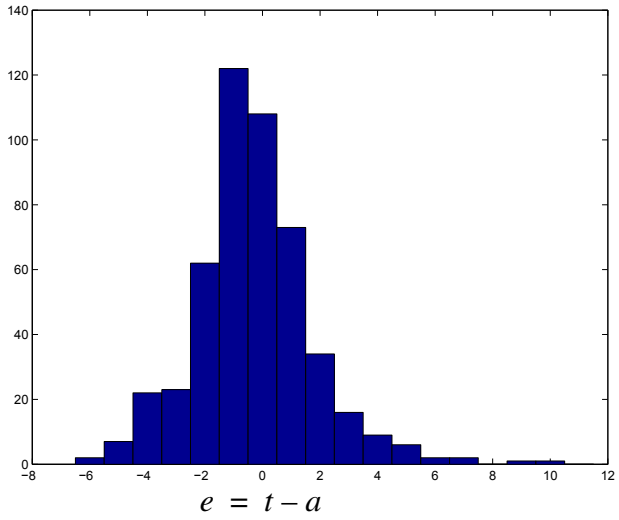

Figure 22.7 Histogram of Network Errors

Pattern Recognition

Confusion Matrix

For pattern recognition problems, the regression analysis is not as useful as it is for fitting problems, since the target values are discrete. However, there is an analogous tool - the *confusion (or misclassification) matrix*. The confusion matrix is a table whose columns represent the target class and whose rows represent the output class. For example, Figure 22.8 shows a sample confusion matrix in which there were 214 data points. There were 41 input vectors that belonged to Class 1 and were correctly classified as Class 1. There were 162 input vectors that belonged to Class 2 and were correctly classified as Class 2. The correctly classified inputs show in the diagonal cells of the confusion matrix. The off-diagonal cells show misclassified inputs. The lower left cell shows that four inputs from Class 1 were misclassified by the network as Class 2. If Class 1 is considered a positive outcome, then the lower left cell represents *false negatives*, which are also

False Negative

False Positive

called Type II errors. The upper right cell shows that one input from Class 2 was misclassified by the network as Class 1. This would be considered a *false positive* or a Type I error.

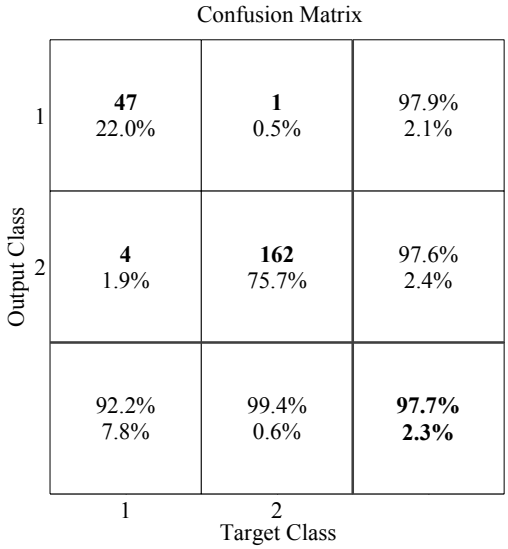

Figure 22.8 Sample Confusion Matrix

ROC Curve

Another useful tool for analyzing a pattern recognition network is called the *Receiver Operating Characteristic (ROC) curve*. To create this curve, we take the output of the trained network and compare it against a threshold which ranges from -1 to +1 (assuming a tansig transfer function in the last layer). Inputs that produce values above the threshold are considered to belong to Class 1, and those with values below the threshold are considered to belong to Class 2. For each threshold value, we count the fraction of true positives and false positives in the data set. This pair of numbers produces one point on the ROC curve. As the threshold is varied, we trace the complete curve, as shown in Figure 22.9.

The ideal point for the ROC curve to pass through would be (0,1), which would correspond to no false positives and all true positives. A poor ROC curve would represent a random guess, which is represented by the diagonal line in Figure 22.9, which passes through the point (0.5,0.5).

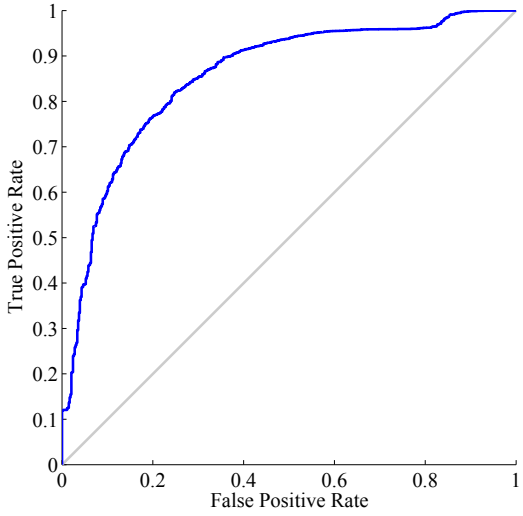

Figure 22.9 Receiver Operating Characteristic Curve

Clustering

Quantization Error

The SOM is the most commonly used network for clustering. There are several measures of SOM performance. One is *quantization error*. This is the average distance between each input vector and the closest prototype vector. It measures the map resolution. This can be made artificially small, if we use a large number of neurons. If there are as many neurons as input vectors in the data set, then the quantization error could be zero. This would represent overfitting. If the number of neurons is not significantly smaller than the number of input vectors, then the quantization error is not meaningful.

Topographic Error

Another measure of SOM performance is *topographic error*. This is the proportion of all input vectors for which the closest prototype vector and the next closest prototype vector are not neighbors in the feature map topology. Topographic error measures the preservation of the topology. In a well-trained SOM, prototypes that are neighbors in the topology should also be neighbors in the input space. In this case, the topographic error should be zero.

Distortion Measure

The performance of the SOM can also be assessed by the *distortion measure*:

$$E_d = \sum_{q=1}^{Q} \sum_{i=1}^{S} h_{ic_q} \left\| {}_i\mathbf{w} - \mathbf{p}_q \right\|^2, \tag{22.14}$$

where h_{ij} is the neighborhood function, and c_q is the index of the prototype that is closest to the input vector $\mathbf{p}_q$:

$$c_q = \arg\min_j \{ \| {}_j\mathbf{w} - \mathbf{p}_q \| \} . \tag{22.15}$$

For the simplest neighborhood function, h_{ij} is equal to 1 if prototype i is within some pre-specified neighborhood radius of prototype j, and equal to zero otherwise. It is also possible to have neighborhood functions that decrease continuously, such as the Gaussian function:

$$h_{ij} = \exp\left(\frac{-\| {}_i\mathbf{w} - {}_j\mathbf{w} \|^2}{2d^2} \right), \tag{22.16}$$

where d is the neighborhood radius.

Prediction

As we discussed earlier, one application of neural networks is the prediction of the future values of some time series. For prediction problems, we use dynamic networks, such as the focused time-delay neural network shown in Figure 22.3. There are two important concepts that are used when analyzing a trained prediction network:

1. the prediction errors should not be correlated in time, and
2. the prediction errors should not be correlated with the input sequence.

If the prediction errors were correlated in time, then we would be able to predict the prediction errors and, therefore, improve our original prediction. Also, if the prediction errors were correlated with the input sequence, then we would also be able to use this correlation to predict the errors.

Autocorrelation Function

In order to test the correlation of the prediction errors in time, we can use the sample *autocorrelation function*:

$$R_e(\tau) = \frac{1}{Q-\tau} \sum_{t=1}^{Q-\tau} e(t)e(t+\tau) . \tag{22.17}$$

White Noise

If the prediction errors are uncorrelated (*white noise*), then we would expect $R_e(\tau)$ to be close to zero, except when $\tau = 0$. To determine if $R_e(\tau)$ is close to zero, we can set an approximate 95% confidence interval [BoJe96] using the range

$$-\frac{2R_e(0)}{\sqrt{Q}} < R_e(\tau) < \frac{2R_e(0)}{\sqrt{Q}} . \tag{22.18}$$

We can say that $e(t)$ is white, if $R_e(\tau)$ satisfies Eq. (22.18) for $\tau \neq 0$. This concept is illustrated in Figure 22.10 and Figure 22.11. Figure 22.10 shows a sample autocorrelation function for the prediction errors of a network that has not been adequately trained. We can see that the autocorrelation

function does not fall totally within the bounds defined by Eq. (22.18), which are indicated by the dashed lines in the figure. Figure 22.11 shows the corresponding autocorrelation function when a network has been successfully trained. $R_e(\tau)$ falls within the bounds, except at $\tau = 0$.

Correlation in the prediction errors can indicate that the length of the tapped delay lines in the network should be increased.

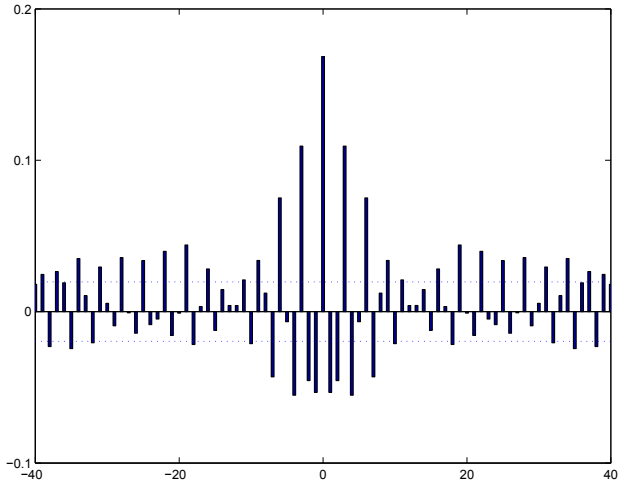

Figure 22.10 $R_e(\tau)$ for Inadequately Trained Network

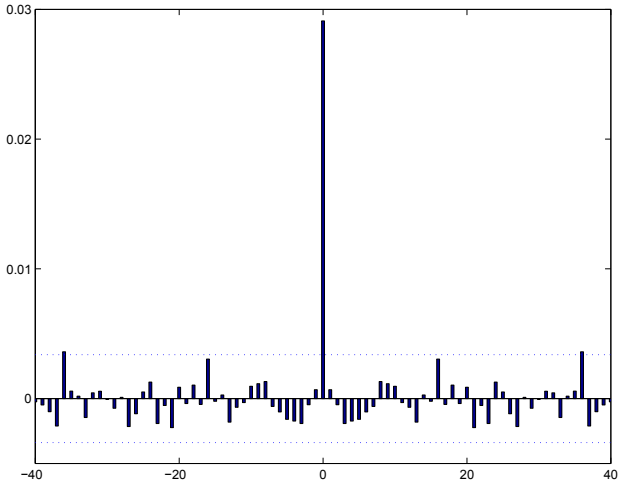

Figure 22.11 $R_e(\tau)$ for Successfully Trained Network

Cross-correlation Function

To test the correlation between the prediction errors and the input sequence, we can use the sample *cross-correlation function*:

$$R_{pe}(\tau) = \frac{1}{Q-\tau}\sum_{t=1}^{Q-\tau} p(t)e(t+\tau). \tag{22.19}$$

If there is no correlation between the prediction errors and the input sequence, then we would expect $R_{pe}(\tau)$ to be close to zero for all τ. To determine if $R_{pe}(\tau)$ is close to zero, we can set an approximate 95% confidence interval [BoJe96] using the range

$$-\frac{2\sqrt{R_e(0)}\sqrt{R_p(0)}}{\sqrt{Q}} < R_{pe}(\tau) < \frac{2\sqrt{R_e(0)}\sqrt{R_p(0)}}{\sqrt{Q}}. \tag{22.20}$$

This concept is illustrated in Figure 22.12 and Figure 22.13. Figure 22.12 shows a sample cross-correlation function for the prediction errors of a network that has not been adequately trained. We can see that the cross-correlation function does not fall totally within the bounds defined by Eq. (22.20), which are indicated by the dashed lines. Figure 22.13 shows the corresponding cross-correlation function when a network has been successfully trained. $R_{pe}(\tau)$ falls within the bounds for all τ.

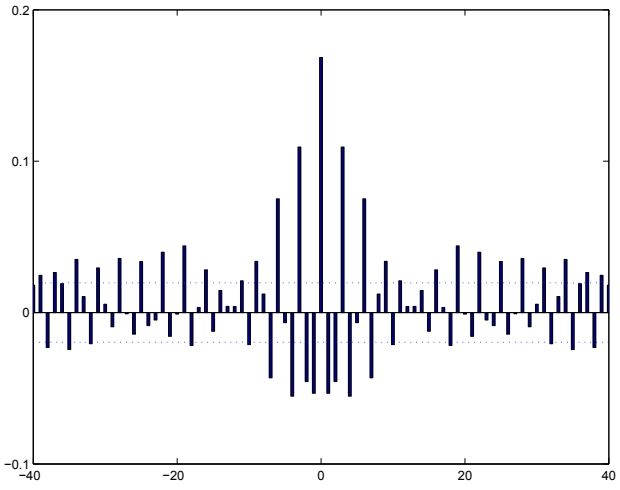

Figure 22.12 $R_{pe}(\tau)$ for Inadequately Trained Network

When using a NARX network, correlation between the prediction error and the input can suggest that the lengths of the tapped delay lines in the input and feedback paths should be increased.

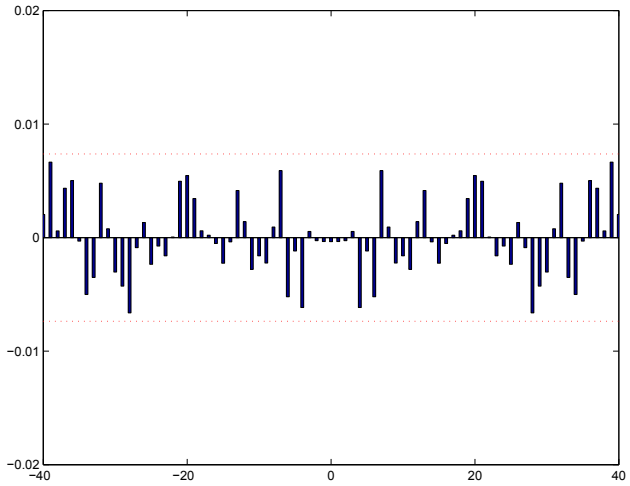

Figure 22.13 $R_{pe}(\tau)$ for Successfully Trained Network

Overfitting and Extrapolation

Recall from Chapter 13 that the total data set is divided into three parts: training, validation and testing. The training set is used to calculate gradients and to determine weight updates. The validation set is used to stop training before overfitting occurs. (If Bayesian regularization is used, then the validation set may be merged with the training set.) The test set is used to predict future performance of the network. The test set performance is the measure of network quality. If, after a network has been trained, the test set performance is not adequate, then there are usually four possible causes:

- the network has reached a local minimum,
- the network does not have enough neurons to fit the data,
- the network is overfitting, or
- the network is extrapolating.

The local minimum problem can almost always be overcome by retraining the network with five to ten random sets of initial weights. The network with minimum training error will generally represent a global minimum. The other three problems can generally be distinguished by analyzing the training, validation and test set errors. For example, if the validation error is much larger than the training error, then overfitting has probably occurred. Even though early stopping is used, it is possible to have some overfitting, if the training occurs too quickly. In this case, we can use a slower training algorithm to retrain the network.

If the validation, training and test errors are all similar in size, but the errors are too large, then it is likely that the network is not powerful enough to fit the data. In this case, we should increase the number of neurons in the hidden layer and retrain the network. If Bayesian regularization is used, this situation is indicated by the effective number of parameters becoming equal to the total number of parameters. When the network is large enough, the effective number of parameters should remain below the total number of parameters.

If the validation and training errors are similar in size, but the test errors are significantly larger, then the network may be extrapolating. This indicates that the test data fall outside the range of the training and validation data. In this case, we need to get more data. You can merge the test data into the training/validation data and then collect new test data. You should continue to add data until the results on all three data sets are similar.

If training, validation and test errors are similar, and the errors are small enough, then we can put the multilayer network to use. However, we still need to be careful about the possibility of extrapolation. If the multilayer network inputs are outside the range of the data with which it was trained, then extrapolation will occur. It is difficult to guarantee that training data will encompass all future uses of a neural network.

One method for detecting extrapolation is to train a companion competitive network to cluster the input vectors in the multilayer network training set. Then, when an input is applied to the multilayer network, the same input is applied to the companion competitive network. When the distance of the input vector to the nearest prototype vector of the competitive network is larger than the distance from the prototype to the most distant member of its cluster of inputs in the training set, we can suspect extrapolation. This technique is referred to as *novelty detection*.

Novelty Detection

Sensitivity Analysis

After a multilayer network has been trained, it is often useful to assess the importance of each element of the input vector. If we can determine that a given element of the input vector is unimportant, then we can eliminate it. This can simplify the network, reduce the amount of computation and help prevent overfitting. There is no one method that can absolutely determine the importance of each input, but a sensitivity analysis can be helpful in this regard. A sensitivity analysis computes the derivatives of the network response with respect to each element of the input vector. If the derivative with respect to a certain input element is small, then that element can be eliminated from the input vector.

Because the multilayer network is nonlinear, the derivative of the network output with respect to an input element will not be constant. For each input vector in the training set, the derivatives will be different. For this reason, we can't use a single derivative to determine sensitivity. One option would

be to take the average of the absolute derivatives, or else the rms derivatives, over the entire training set. Another option would be to compute the derivative of the sum square error with respect to each element of the input vector. Each of these will compute a single derivative for each element of the input vector. The last computation can be performed with a simple variation of the backpropagation algorithm (see Eq. (11.44) to Eq. (11.47)). Recall from Eq. (11.32) that

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}, \tag{22.21}$$

where $\hat{F}$ is a single error squared. We want to convert this to a derivative with respect to an element of the input vector, using the chain rule:

$$\frac{\partial \hat{F}}{\partial p_j} = \sum_{i=1}^{S^1} \frac{\partial \hat{F}}{\partial n_i^1} \times \frac{\partial n_i^1}{\partial p_j} = \sum_{i=1}^{S^1} s_i^1 \times \frac{\partial n_i^1}{\partial p_j}. \tag{22.22}$$

We know that

$$n_i^1 = \sum_{j=1}^{R} w_{i,j}^1 p_j + b_i^1, \tag{22.23}$$

therefore, Eq. (22.22) becomes

$$\frac{\partial \hat{F}}{\partial p_j} = \sum_{i=1}^{S^1} \frac{\partial \hat{F}}{\partial n_i^1} \times \frac{\partial n_i^1}{\partial p_j} = \sum_{i=1}^{S^1} s_i^1 \times w_{i,j}^1. \tag{22.24}$$

In matrix form, we can write this as

$$\frac{\partial \hat{F}}{\partial \mathbf{p}} = (\mathbf{W}^1)^T \mathbf{s}^1. \tag{22.25}$$

This will be the derivative for a single squared error. To get the derivative of the sum square error, we sum the individual derivatives for each single squared error. The resulting vector will contain the derivatives of the sum square error for each element of the input vector. If we find that some of these derivatives are much smaller than the maximum derivative, then we can consider removing those inputs. After removing the potentially irrelevant inputs, we retrain the network and compare the performance with the original network. If the performance is similar, then we accept the simplified network.

Epilogue

While previous chapters have focused on the fundamentals of particular network architectures and training rules, this chapter has discussed some practical aspects of neural network training. Neural network training is an iterative process involving data collection and preprocessing, network architecture selection, network training and post-training analysis.

The next five chapters will demonstrate some of these practical aspects, as we present some real-world case studies. The case studies will cover a variety of applications, including function fitting, density estimation, pattern recognition, clustering and prediction.

Further Reading

[Bish95] C.M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press,1995.

This well-written and well-organized textbook presents neural networks from a statistical perspective.

[BoJe94] G.E.P. Box, G.M. Jenkins, and G.C. Reinsel, *Time Series Analysis: Forecasting and Control*, 4th Edition, John Wiley & Sons, 2008.

This is a classic text on time series analysis. It focuses on practical aspects, rather than theoretical derivations.

[HaBo07] L. Hamm, B. W. Brorsen and M. T. Hagan, "Comparison of Stochastic Global Optimization Methods to Estimate Neural Network Weights," *Neural Processing Letters*, Vol. 26, No. 3, December 2007.

This paper demonstrates that using multiple restarts of a local optimization procedure, like steepest descent or conjugate gradient, produces results that are comparable to global optimization methods, but with less computation.

[HeOh97] B. Hedén, H. Öhlin, R. Rittner, L. Edenbrandt, "Acute Myocardial Infarction Detected in the 12-Lead ECG by Artificial Neural Networks," *Circulation*, vol. 96, pp. 1798–1802, 1997.

Describes the use of neural networks in detecting myocardial infarctions, using the electrocardiogram.

[Joll02] I.T. Jolliffe, *Principal Component Analysis*, Springer Series in Statistics, 2nd ed., Springer, NY, 2002.

The most popular text on principal component analysis.

[LeCu98] Y. LeCun, L. Bottou, G. B. Orr, K.-R. Mueller, "Efficient BackProp," *Lecture Notes in Comp. Sci.*, vol. 1524, 1998.

This paper presents practical tips that improve the training of multilayer networks.

[Moll93] M. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, pp. 525-533, 1993.

The scaled conjugate gradient algorithm presented in this paper converges quickly, and with a minimum amount of memory requirements.

[NgWi90] D. Nguyen and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the IJCNN*, vol. 3, pp. 21–26, July 1990.

This paper describes a procedure for setting the initial weights and biases for the backpropagation algorithm. It uses the shape of the sigmoid transfer function and the range of the input variables to determine how large the weights should be, and then uses the biases to center the sigmoids in the operating region. The convergence of backpropagation is improved significantly by this procedure.

[PeCo93] M. P. Perrone and L. N. Cooper, "When networks disagree: Ensemble methods for hybrid neural networks," in *Neural Networks for Speech and Image Processing*, R. J. Mammone, Ed., Chapman-Hall, pp. 126-142, 1993.

This paper describes how you can combine the outputs of a committee of networks to produce results that are more accurate than any of the individual networks.

[PuFe97] G.V. Puskorius and L.A. Feldkamp, "Extensions and enhancements of decoupled extended Kalman filter training," *Proceedings of the 1997 International Conference on Neural Networks*, vol. 3, pp. 1879-1883, 1997.

The extended Kalman filter algorithm described in this paper is one of the faster sequential algorithms for neural network training.

[RaMa05] L.M. Raff, M. Malshe, M. Hagan, D.I. Doughan, M.G. Rockley, and R. Komanduri, "*Ab initio* potential-energy surfaces for complex, multi-channel systems using modified novelty sampling and feedforward neural networks," *The Journal of Chemical Physics*, vol. 122, 2005.

This paper describes how neural networks can be used to model molecular interactions.

[ScSm99] B. Schölkopf, A. Smola, K.-R. Muller, "Kernel Principal Component Analysis," in B. Schölkopf, C. J. C. Burges, A. J. Smola (Eds.), *Advances in Kernel Methods-Support Vector Learning*, MIT Press Cambridge, MA, USA, pp. 327-352, 1999.

This paper introduces a nonlinear version of principal component analysis using a kernel method.