www.statsoft.com

- Products
- Solutions
- Buy
- Trials
- Support

TextbookNeural Networks

What can we help you find? | Search

- Elementary Concepts
- Statistics Glossary
- Basic Statistics
- ANOVA / MANOVA
- Association Rules
- Boosting Trees
- Canonical Analysis
- CHAID Analysis
- C & R Trees
- Classification Trees
- Cluster Analysis
- Correspondence Analysis
- Data Mining Techniques
- Discriminant Analysis
- Distribution Fitting
- Experimental Design
- Factor Analysis
- General Discrim. Analysis
- General Linear Models
- Generalized Additive Mod.
- Generalized Linear Mod.
- General Regression Mod.
- Graphical Techniques
- Ind.Components Analysis
- Linear Regression
- Log-Linear Analysis
- MARSplines
- Machine Learning
- Multidimensional Scaling
- Neural Networks
- Nonlinear Estimation
- Nonparametric Statistics
- Partial Least Squares
- Power Analysis
- Process Analysis
- Quality Control Charts
- Reliability / Item Analysis
- SEPATH (Structural eq.)
- Survival Analysis
- Text Mining
- Time Series / Forecasting

Model Extremely Complex Functions, Neural Networks

Data Mining, Neural Networks for Regression - Session...

▶

- Preface
- Applications for Neural Networks
- The Biological Inspiration
- The Basic Artificial Model
- Using a Neural Network
- Gathering Data for Neural Networks
  - Summary
- Pre- and Post-processing
- Multilayer Perceptrons
  - Training Multilayer Perceptrons
  - The Back Propagation Algorithm
  - Over-Learning and Generalization
  - Data Selection

- - Insights into MLP Training
  - Other MLP Training Algorithms
- Radial Basis Function Networks
- SOFM Networks
- Classification in Neural Networks
  - Classification Statistics
- Regression Problems in Neural Networks
- Time Series Prediction in Neural Networks
- Variable Selection and Dimensionality Reduction
- Ensembles and Resampling
- Recommended Textbooks

Many concepts related to the neural networks methodology are best explained if they are illustrated with applications of a specific neural network program. Therefore, this section contains many references to *STATISTICA Neural Networks*, a particularly comprehensive neural networks application available from StatSoft.

Preface

Neural networks have seen an explosion of interest over the last few years, and are being successfully applied across an extraordinary range of problem domains, in areas as diverse as finance, medicine, engineering, geology and physics. Indeed, anywhere that there are problems of prediction, classification or control, neural networks are being introduced. This sweeping success can be attributed to a few key factors:

- **Power.** Neural networks are very sophisticated modeling techniques capable of modeling extremely complex functions. In particular, neural networks are *nonlinear* (a term which is discussed in more detail later in this section). For many years linear modeling has been the commonly used technique in most modeling domains since linear models have well-known optimization strategies. Where the linear approximation was not valid (which was frequently the case) the models suffered accordingly. Neural networks also keep in check the *curse of dimensionality* problem that bedevils attempts to model nonlinear functions with large numbers of variables.
- **Ease of use.** Neural networks *learn by example*. The neural network user gathers representative data, and then invokes *training algorithms* to automatically learn the structure of the data. Although the user does need to have some heuristic knowledge of how to select and prepare data, how to select an appropriate neural network, and how to interpret the results, the level of user knowledge needed to successfully apply neural networks is much lower than would be the case using (for example) some more traditional nonlinear statistical methods.

Neural networks are also intuitively appealing, based as they are on a crude low-level model of biological neural systems. In the future, the development of this neurobiological modeling may lead to genuinely intelligent computers.

Applications for Neural Networks

Neural networks are applicable in virtually every situation in which a relationship between the predictor variables (independents, inputs) and predicted variables (dependents, outputs) exists, even when that relationship is very complex and not easy to articulate in the usual terms of "correlations" or "differences between groups." A few representative examples of problems to

which neural network analysis has been applied successfully are:

- **Detection of medical phenomena.** A variety of health-related indices (e.g., a combination of heart rate, levels of various substances in the blood, respiration rate) can be monitored. The onset of a particular medical condition could be associated with a very complex (e.g., nonlinear and interactive) combination of changes on a subset of the variables being monitored. Neural networks have been used to recognize this predictive pattern so that the appropriate treatment can be prescribed.

- **Stock market prediction.** Fluctuations of stock prices and stock indices are another example of a complex, multidimensional, but in some circumstances at least partially-deterministic phenomenon. Neural networks are being used by many technical analysts to make predictions about stock prices based upon a large number of factors such as past performance of other stocks and various economic indicators.

- **Credit assignment.** A variety of pieces of information are usually known about an applicant for a loan. For instance, the applicant's age, education, occupation, and many other facts may be available. After training a neural network on historical data, neural network analysis can identify the most relevant characteristics and use those to classify applicants as good or bad credit risks.

- **Monitoring the condition of machinery.** Neural networks can be instrumental in cutting costs by bringing additional expertise to scheduling the preventive maintenance of machines. A neural network can be trained to distinguish between the sounds a machine makes when it is running normally ("false alarms") versus when it is on the verge of a problem. After this training period, the expertise of the network can be used to warn a technician of an upcoming breakdown, before it occurs and causes costly unforeseen "downtime."

- **Engine management.** Neural networks have been used to analyze the input of sensors from an engine. The neural network controls the various parameters within which the engine functions, in order to achieve a particular goal, such as minimizing fuel consumption.

The Biological Inspiration

Neural networks grew out of research in Artificial Intelligence; specifically, attempts to mimic the fault-tolerance and capacity to learn of biological neural systems by modeling the low-level structure of the brain (see Patterson, 1996). The main branch of Artificial Intelligence research in the 1960s -1980s produced Expert Systems. These are based upon a high-level model of reasoning processes (specifically, the concept that our reasoning processes are built upon manipulation of symbols). It became rapidly apparent that these systems, although very useful in some domains, failed to capture certain key aspects of human intelligence. According to one line of speculation, this was due to their failure to mimic the underlying structure of the brain. In order to reproduce intelligence, it would be necessary to build systems with a similar architecture.

The brain is principally composed of a very large number (circa 10,000,000,000) of *neurons*, massively interconnected (with an average of several thousand interconnects per neuron, although this varies enormously). Each neuron is a specialized cell which can propagate an electrochemical signal. The neuron has a branching input structure (the dendrites), a cell body, and a branching output structure (the axon). The axons of one cell connect to the dendrites of another via a synapse. When a neuron is activated, it *fires* an electrochemical signal along the axon. This signal crosses the synapses to other neurons, which may in turn fire. A neuron fires only if the total signal received at the cell body from the dendrites exceeds a certain level (the firing threshold).

The strength of the signal received by a neuron (and therefore its chances of firing) critically

depends on the efficacy of the synapses. Each synapse actually contains a gap, with neurotransmitter chemicals poised to transmit a signal across the gap. One of the most influential researchers into neurological systems (Donald Hebb) postulated that learning consisted principally in altering the "strength" of synaptic connections. For example, in the classic Pavlovian conditioning experiment, where a bell is rung just before dinner is delivered to a dog, the dog rapidly learns to associate the ringing of a bell with the eating of food. The synaptic connections between the appropriate part of the auditory cortex and the salivation glands are strengthened, so that when the auditory cortex is stimulated by the sound of the bell the dog starts to salivate. Recent research in cognitive science, in particular in the area of nonconscious information processing, have further demonstrated the enormous capacity of the human mind to infer ("learn") simple input-output covariations from extremely complex stimuli (e.g., see Lewicki, Hill, and Czyzewska, 1992).

Thus, from a very large number of extremely simple processing units (each performing a weighted sum of its inputs, and then firing a binary signal if the total input exceeds a certain level) the brain manages to perform extremely complex tasks. Of course, there is a great deal of complexity in the brain which has not been discussed here, but it is interesting that artificial neural networks can achieve some remarkable results using a model not much more complex than this.

The Basic Artificial Model

To capture the essence of biological neural systems, an artificial *neuron* is defined as follows:

- It receives a number of inputs (either from original data, or from the output of other neurons in the neural network). Each input comes via a connection that has a strength (or *weight*); these weights correspond to synaptic efficacy in a biological neuron. Each neuron also has a single threshold value. The weighted sum of the inputs is formed, and the threshold subtracted, to compose the *activation* of the neuron (also known as the post-synaptic potential, or PSP, of the neuron).
- The activation signal is passed through an activation function (also known as a transfer function) to produce the output of the neuron.

If the step activation function is used (i.e., the neuron's output is 0 if the input is less than zero, and 1 if the input is greater than or equal to 0) then the neuron acts just like the biological neuron described earlier (subtracting the threshold from the weighted sum and comparing with zero is equivalent to comparing the weighted sum to the threshold). Actually, the step function is rarely used in artificial neural networks, as will be discussed. Note also that weights can be negative, which implies that the synapse has an inhibitory rather than excitatory effect on the neuron: inhibitory neurons are found in the brain.

This describes an individual neuron. The next question is: how should neurons be connected together? If a network is to be of any use, there must be inputs (which carry the values of variables of interest in the outside world) and outputs (which form predictions, or control signals). Inputs and outputs correspond to sensory and motor nerves such as those coming from the eyes and leading to the hands. However, there also can be hidden neurons that play an internal role in the network. The input, hidden and output neurons need to be connected together.

The key issue here is *feedback* (Haykin, 1994). A simple network has a *feedforward* structure: signals flow from inputs, forwards through any hidden units, eventually reaching the output units. Such a structure has stable behavior. However, if the network is *recurrent* (contains connections

back from later to earlier neurons) it can be unstable, and has very complex dynamics. Recurrent networks are very interesting to researchers in neural networks, but so far it is the feedforward structures that have proved most useful in solving real problems.

A typical feedforward network has neurons arranged in a distinct layered topology. The input layer is not really neural at all: these units simply serve to introduce the values of the input variables. The hidden and output layer neurons are each connected to all of the units in the preceding layer. Again, it is possible to define networks that are partially-connected to only some units in the preceding layer; however, for most applications fully-connected networks are better.

When the network is executed (used), the input variable values are placed in the input units, and then the hidden and output layer units are progressively executed. Each of them calculates its activation value by taking the weighted sum of the outputs of the units in the preceding layer, and subtracting the threshold. The activation value is passed through the activation function to produce the output of the neuron. When the entire network has been executed, the outputs of the output layer act as the output of the entire network.

To index

Using a Neural Network

The previous section describes in simplified terms how a neural network turns inputs into outputs. The next important question is: how do you apply a neural network to solve a problem?

The type of problem amenable to solution by a neural network is defined by the way they *work* and the way they are *trained*. Neural networks work by feeding in some input variables, and producing some output variables. They can therefore be used where you have some known information, and would like to infer some unknown information (see Patterson, 1996; Fausett, 1994). Some examples are:

Stock market prediction. You know last week's stock prices and today's DOW, NASDAQ, or FTSE index; you want to know tomorrow's stock prices.

Credit assignment. You want to know whether an applicant for a loan is a good or bad credit risk. You usually know applicants' income, previous credit history, etc. (because you ask them these things).

Control. You want to know whether a robot should turn left, turn right, or move forwards in order to reach a target; you know the scene that the robot's camera is currently observing.

Needless to say, not every problem can be solved by a neural network. You may wish to know next week's lottery result, and know your shoe size, but there is no relationship between the two. Indeed, if the lottery is being run correctly, there is no fact you could possibly know that would allow you to infer next week's result. Many financial institutions use, or have experimented with, neural networks for stock market prediction, so it is likely that any trends predictable by neural techniques are already discounted by the market, and (unfortunately), unless you have a sophisticated understanding of that problem domain, you are unlikely to have any success there either.

Therefore, another important requirement for the use of a neural network therefore is that you know (or at least strongly suspect) that there is a relationship between the proposed known inputs and unknown outputs. This relationship may be noisy (you certainly would not expect that the factors given in the stock market prediction example above could give an exact prediction, as prices are clearly influenced by other factors not represented in the input set, and there may be an element of pure randomness) but it must exist.

In general, if you use a neural network, you won't know the exact nature of the relationship

between inputs and outputs - if you knew the relationship, you would model it directly. The other key feature of neural networks is that they learn the input/output relationship through training. There are two types of training used in neural networks, with different types of networks using different types of training. These are supervised and unsupervised training, of which supervised is the most common and will be discussed in this section (unsupervised learning is described in a later section).

In supervised learning, the network user assembles a set of *training data*. The training data contains examples of inputs together with the corresponding outputs, and the network learns to infer the relationship between the two. Training data is usually taken from historical records. In the above examples, this might include previous stock prices and DOW, NASDAQ, or FTSE indices, records of previous successful loan applicants, including questionnaires and a record of whether they defaulted or not, or sample robot positions and the correct reaction.

The neural network is then trained using one of the supervised learning algorithms (of which the best known example is *back propagation*, devised by Rumelhart et. al., 1986), which uses the data to adjust the network's weights and thresholds so as to minimize the error in its predictions on the training set. If the network is properly trained, it has then learned to model the (unknown) function that relates the input variables to the output variables, and can subsequently be used to make predictions where the output is *not* known.

Gathering Data for Neural Networks

Once you have decided on a problem to solve using neural networks, you will need to gather data for training purposes. The training data set includes a number of *cases*, each containing values for a range of input and output *variables*. The first decisions you will need to make are: which variables to use, and how many (and which) cases to gather.

The choice of variables (at least initially) is guided by intuition. Your own expertise in the problem domain will give you some idea of which input variables are likely to be influential. As a first pass, you should include any variables that you think could have an influence - part of the design process will be to whittle this set down.

Neural networks process numeric data in a fairly limited range. This presents a problem if data is in an unusual range, if there is missing data, or if data is non-numeric. Fortunately, there are methods to deal with each of these problems. Numeric data is scaled into an appropriate range for the network, and missing values can be substituted for using the mean value (or other statistic) of that variable across the other available training cases (see Bishop, 1995).

Handling non-numeric data is more difficult. The most common form of non-numeric data consists of nominal-value variables such as *Gender*={*Male, Female*}. Nominal-valued variables can be represented numerically. However, neural networks do not tend to perform well with nominal variables that have a large number of possible values.

For example, consider a neural network being trained to estimate the value of houses. The price of houses depends critically on the area of a city in which they are located. A particular city might be subdivided into dozens of named locations, and so it might seem natural to use a nominal-valued variable representing these locations. Unfortunately, it would be very difficult to train a neural network under these circumstances, and a more credible approach would be to assign ratings (based on expert knowledge) to each area; for example, you might assign ratings for the quality of local schools, convenient access to leisure facilities, etc.

Other kinds of non-numeric data must either be converted to numeric form, or discarded. Dates and times, if important, can be converted to an offset value from a starting date/time. Currency

values can easily be converted. Unconstrained text fields (such as names) cannot be handled and should be discarded.

The number of cases required for neural network training frequently presents difficulties. There are some heuristic guidelines, which relate the number of cases needed to the size of the network (the simplest of these says that there should be ten times as many cases as connections in the network). Actually, the number needed is also related to the (unknown) complexity of the underlying function which the network is trying to model, and to the variance of the additive noise. As the number of variables increases, the number of cases required increases nonlinearly, so that with even a fairly small number of variables (perhaps fifty or less) a huge number of cases are required. This problem is known as "the curse of dimensionality," and is discussed further later in this section.

For most practical problem domains, the number of cases required will be hundreds or thousands. For very complex problems more may be required, but it would be a rare (even trivial) problem which required less than a hundred cases. If your data is sparser than this, you really don't have enough information to train a network, and the best you can do is probably to fit a linear model. If you have a larger, but still restricted, data set, you can compensate to some extent by forming an ensemble of networks, each trained using a different resampling of the available data, and then average across the predictions of the networks in the ensemble.

Many practical problems suffer from data that is unreliable: some variables may be corrupted by noise, or values may be missing altogether. Neural networks are also noise tolerant. However, there is a limit to this tolerance; if there are occasional outliers far outside the range of normal values for a variable, they may bias the training. The best approach to such outliers is to identify and remove them (either discarding the case, or converting the outlier into a missing value).

SUMMARY

Choose variables that you believe may be influential.

Numeric and nominal variables can be handled. Convert other variables to one of these forms, or discard.

Hundreds or thousands of cases are required; the more variables, the more cases.

Cases with missing values can be used, if necessary, but outliers may cause problems - check your data. Remove outliers if possible. If you have sufficient data, discard cases with missing values.

If the volume of the data available is small, consider using ensembles and resampling.

Pre- and Post-Processing

All neural networks take numeric input and produce numeric output. The transfer function of a unit is typically chosen so that it can accept *input in any* range, and produces *output in a strictly limited* range (it has a squashing effect). Although the input can be in any range, there is a saturation effect so that the unit is only sensitive to inputs within a fairly limited range. The illustration below shows one of the most common transfer functions, the logistic function (also sometimes referred to as the sigmoid function, although strictly speaking it is only one example of a sigmoid - S-shaped - function). In this case, the output is in the range (0,1), and the input is sensitive in a range not much larger than (-1,+1). The function is also smooth and easily differentiable, facts that are critical in allowing the network training algorithms to operate (this is the reason why the step function is not used in practice).
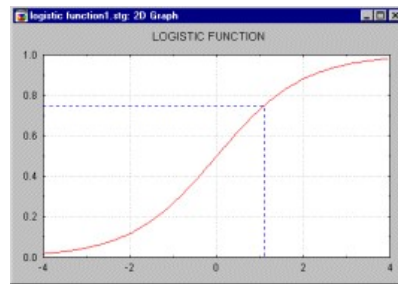
The limited numeric response range, together with the fact that information has to be in numeric form, implies that neural solutions require preprocessing and post-processing stages to be used in real applications (see Bishop, 1995). Two issues need to be addressed:

Scaling. Numeric values have to be scaled into a range that is appropriate for the network. Typically, raw variable values are scaled linearly. In some circumstances, non-linear scaling may be appropriate (for example, if you know that a variable is exponentially distributed, you might take the logarithm). Non-linear scaling is not supported in *ST Neural Networks*. Instead, you should scale the variable using *STATISTICA*'s data transformation facilities before transferring the data to *ST Neural Networks*.

Nominal variables. Nominal variables may be two-state (e.g., *Gender*={*Male*,*Female*}) or many-state (i.e., more than two states). A two-state nominal variable is easily represented by transformation into a numeric value (e.g., *Male=0, Female=1*). Many-state nominal variables are more difficult to handle. They can be represented using an ordinal encoding (e.g., *Dog=0,Budgie=1,Cat=2*) but this implies a (probably) false ordering on the nominal values - in this case, that *Budgies* are in some sense midway between *Dogs* and *Cats*. A better approach, known as *one-of-N* encoding, is to use a number of numeric variables to represent the single nominal variable. The number of numeric variables equals the number of possible values; one of the *N* variables is set, and the others cleared (e.g., *Dog={1,0,0}*, *Budgie={0,1,0}*, *Cat={0,0,1}*). *ST Neural Networks* has facilities to convert both two-state and many-state nominal variables for use in the neural network. Unfortunately, a nominal variable with a large number of states would require a prohibitive number of numeric variables for *one-of-N* encoding, driving up the network size and making training difficult. In such a case it is possible (although unsatisfactory) to model the nominal variable using a single numeric ordinal; a better approach is to look for a different way to represent the information.

Prediction problems may be divided into two main categories:

Classification. In classification, the objective is to determine to which of a number of discrete classes a given input case belongs. Examples include credit assignment (is this person a good or bad credit risk), cancer detection (tumor, clear), signature recognition (forgery, true). In all these cases, the output required is clearly a single nominal variable. The most common classification tasks are (as above) two-state, although many-state tasks are also not unknown.

Regression. In regression, the objective is to predict the value of a (usually) continuous variable: tomorrow's stock price, the fuel consumption of a car, next year's profits. In this case, the output required is a single numeric variable.

Neural networks can actually perform a number of regression and/or classification tasks at once, although commonly each network performs only one. In the vast majority of cases, therefore, the network will have a single output variable, although in the case of many-state classification problems, this may correspond to a number of output units (the post-processing stage takes care of the mapping from output units to output variables). If you do define a single network with multiple output variables, it may suffer from cross-talk (the hidden neurons experience difficulty learning, as they are attempting to model at least two functions at once). The best solution is

usually to train separate networks for each output, then to combine them into an ensemble so that they can be run as a unit.

Multilayer Perceptrons

This is perhaps the most popular network architecture in use today, due originally to Rumelhart and McClelland (1986) and discussed at length in most neural network textbooks (e.g., Bishop, 1995). This is the type of network discussed briefly in previous sections: the units each perform a biased weighted sum of their inputs and pass this activation level through a transfer function to produce their output, and the units are arranged in a layered feedforward topology. The network thus has a simple interpretation as a form of input-output model, with the weights and thresholds (biases) the free parameters of the model. Such networks can model functions of almost arbitrary complexity, with the number of layers, and the number of units in each layer, determining the function complexity. Important issues in Multilayer Perceptrons (MLP) design include specification of the number of hidden layers and the number of units in these layers (see Haykin, 1994; Bishop, 1995).

The number of input and output units is defined by the problem (there may be some uncertainty about precisely which inputs to use, a point to which we will return later. However, for the moment we will assume that the input variables are intuitively selected and are all meaningful). The number of hidden units to use is far from clear. As good a starting point as any is to use one hidden layer, with the number of units equal to half the sum of the number of input and output units. Again, we will discuss how to choose a sensible number later.

TRAINING MULTILAYER PERCEPTRONS

Once the number of layers, and number of units in each layer, has been selected, the network's weights and thresholds must be set so as to minimize the prediction error made by the network. This is the role of the *training algorithms*. The historical cases that you have gathered are used to automatically adjust the weights and thresholds in order to minimize this error. This process is equivalent to fitting the model represented by the network to the training data available. The error of a particular configuration of the network can be determined by running all the training cases through the network, comparing the actual output generated with the desired or target outputs. The differences are combined together by an *error function* to give the network error. The most common error functions are the *sum squared error* (used for regression problems), where the individual errors of output units on each case are squared and summed together, and the cross entropy functions (used for maximum likelihood classification).

In traditional modeling approaches (e.g., linear modeling) it is possible to algorithmically determine the model configuration that absolutely minimizes this error. The price paid for the greater (non-linear) modeling power of neural networks is that although we can adjust a network to lower its error, we can never be sure that the error could not be lower still.

A helpful concept here is the error surface. Each of the N weights and thresholds of the network (i.e., the free parameters of the model) is taken to be a dimension in space. The $N+1$th dimension is the network error. For any possible configuration of weights the error can be plotted in the $N+1$th dimension, forming an *error surface*. The objective of network training is to find the lowest point in this many-dimensional surface.

In a linear model with sum squared error function, this error surface is a parabola (a quadratic), which means that it is a smooth bowl-shape with a single minimum. It is therefore "easy" to locate the minimum.

Neural network error surfaces are much more complex, and are characterized by a number of

unhelpful features, such as local minima (which are lower than the surrounding terrain, but above the global minimum), flat-spots and plateaus, saddle-points, and long narrow ravines.

It is not possible to analytically determine where the global minimum of the error surface is, and so neural network training is essentially an exploration of the error surface. From an initially random configuration of weights and thresholds (i.e., a random point on the error surface), the training algorithms incrementally seek for the global minimum. Typically, the gradient (slope) of the error surface is calculated at the current point, and used to make a downhill move. Eventually, the algorithm stops in a low point, which may be a local minimum (but hopefully is the global minimum).

THE BACK PROPAGATION ALGORITHM

The best-known example of a neural network training algorithm is *back propagation* (see Patterson, 1996; Haykin, 1994; Fausett, 1994). In back propagation, the gradient vector of the error surface is calculated. This vector points along the line of steepest descent from the current point, so we know that if we move along it a "short" distance, we will decrease the error. A sequence of such moves (slowing as we near the bottom) will eventually find a minimum of some sort. The difficult part is to decide how large the steps should be.

Large steps may converge more quickly, but may also overstep the solution or (if the error surface is very eccentric) go off in the wrong direction. A classic example of this in neural network training is where the algorithm progresses very slowly along a steep, narrow, valley, bouncing from one side across to the other. In contrast, very small steps may go in the correct direction, but they also require a large number of iterations. In practice, the step size is proportional to the slope (so that the algorithms settles down in a minimum) and to a special constant: the *learning rate*. The correct setting for the learning rate is application-dependent, and is typically chosen by experiment; it may also be time-varying, getting smaller as the algorithm progresses.

The algorithm is also usually modified by inclusion of a momentum term: this encourages movement in a fixed direction, so that if several steps are taken in the same direction, the algorithm "picks up speed", which gives it the ability to (sometimes) escape local minimum, and also to move rapidly over flat spots and plateaus.

The algorithm therefore progresses iteratively, through a number of *epochs*. On each epoch, the training cases are each submitted in turn to the network, and target and actual outputs compared and the error calculated. This error, together with the error surface gradient, is used to adjust the weights, and then the process repeats. The initial network configuration is random, and training stops when a given number of epochs elapses or when the error stops improving (you can select which of these stopping conditions to use).

OVER-LEARNING AND GENERALIZATION

One major problem with the approach outlined above is that it doesn't actually minimize the error that we are really interested in - which is the expected error the network will make when *new* cases are submitted to it. In other words, the most desirable property of a network is its ability to *generalize* to new cases. In reality, the network is trained to minimize the error on the training set, and short of having a perfect and infinitely large training set, this is not the same thing as minimizing the error on the real error surface - the error surface of the underlying and unknown model (see Bishop, 1995).

The most important manifestation of this distinction is the problem of over-learning, or over-fitting. It is easiest to demonstrate this concept using polynomial curve fitting rather than neural networks, but the concept is precisely the same.

A polynomial is an equation with terms containing only constants and powers of the variables. For example:

$y=2x+3$
$y=3x^2+4x+1$

Different polynomials have different shapes, with larger powers (and therefore larger numbers of terms) having steadily more eccentric shapes. Given a set of data, we may want to fit a polynomial curve (i.e., a model) to explain the data. The data is probably noisy, so we don't necessarily expect the best model to pass exactly through all the points. A low-order polynomial may not be sufficiently flexible to fit close to the points, whereas a high-order polynomial is actually too flexible, fitting the data exactly by adopting a highly eccentric shape that is actually unrelated to the underlying function. See illustration below.
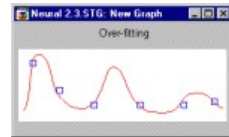
Neural networks have precisely the same problem. A network with more weights models a more complex function, and is therefore prone to over-fitting. A network with less weights may not be sufficiently powerful to model the underlying function. For example, a network with no hidden layers actually models a simple linear function.

How then can we select the right complexity of network? A larger network will almost invariably achieve a lower error eventually, but this may indicate over-fitting rather than good modeling.

The answer is to check progress against an independent data set, the test set. Some of the cases are reserved, and not actually used for training in the *back propagation* algorithm. Instead, they are used to keep an independent check on the progress of the algorithm. It is invariably the case that the initial performance of the network on training and test sets is the same (if it is not at least approximately the same, the division of cases between the two sets is probably biased). As training progresses, the training error naturally drops, and providing training is minimizing the true error function, the selection error drops too. However, if the selection error stops dropping, or indeed starts to rise, this indicates that the network is starting to overfit the data, and training should cease. When over-fitting occurs during the training process like this, it is called over-learning. In this case, it is usually advisable to decrease the number of hidden units and/or hidden layers, as the network is over-powerful for the problem at hand. In contrast, if the network is not sufficiently powerful to model the underlying function, over-learning is not likely to occur, and neither training nor selection errors will drop to a satisfactory level.

The problems associated with local minima, and decisions over the size of network to use, imply that using a neural network typically involves experimenting with a large number of different networks, probably training each one a number of times (to avoid being fooled by local minima), and observing individual performances. The key guide to performance here is the selection error. However, following the standard scientific precept that, all else being equal, a simple model is always preferable to a complex model, you can also select a smaller network in preference to a larger one with a negligible improvement in selection error.

A problem with this approach of repeated experimentation is that the test set plays a key role in selecting the model, which means that it is actually part of the training process. Its reliability as an independent guide to performance of the model is therefore compromised - with sufficient experiments, you may just hit upon a lucky network that happens to perform well on the test set. To add confidence in the performance of the final model, it is therefore normal practice (at least where the volume of training data allows it) to reserve a third set of cases - the validation set. The final model is tested with the validation set data, to ensure that the results on the testing

and training set are real, and not artifacts of the training process. Of course, to fulfill this role properly, the validation set should be used only once - if it is in turn used to adjust and reiterate the training process, it effectively becomes selection data.

This division into multiple subsets is very unfortunate, given that we usually have less data than we would ideally desire even for a single subset. We can get around this problem by resampling. Experiments can be conducted using different divisions of the available data into training, testing, and validation sets. There are a number of approaches to this subset, including random (monte-carlo) resampling and bootstrap. If we make design decisions, such as the best configuration of neural network to use, based upon a number of experiments with different subset examples, the results will be much more reliable. We can then either use those experiments solely to guide the decision as to which network types to use, and train such networks from scratch with new samples (this removes any sampling bias); or, we can retain the best networks found during the sampling process, but average their results in an ensemble, which at least mitigates the sampling bias.

To summarize, network design (once the input variables have been selected) follows a number of stages:

- Select an initial configuration (typically, one hidden layer with the number of hidden units set to half the sum of the number of input and output units).
- Iteratively conduct a number of experiments with each configuration, retaining the best network (in terms of selection error) found. A number of experiments are required with each configuration to avoid being fooled if training locates a local minimum, and it is also best to resample.
- On each experiment, if under-learning occurs (the network doesn't achieve an acceptable performance level) try adding more neurons to the hidden layer(s). If this doesn't help, try adding an extra hidden layer.
- If over-learning occurs (selection error starts to rise) try removing hidden units.
- Once you have experimentally determined an effective configuration for your networks, resample and generate new networks with that configuration.

DATA SELECTION

All the above stages rely on a key assumption. Specifically, the training, verification and test data must be representative of the underlying model (and, further, the three sets must be independently representative). The old computer science adage "garbage in, garbage out" could not apply more strongly than in neural modeling. If training data is not representative, then the model's worth is at best compromised. At worst, it may be useless. It is worth spelling out the kind of problems which can corrupt a training set:

The future is not the past. Training data is typically historical. If circumstances have changed, relationships which held in the past may no longer hold.

All eventualities must be covered. A neural network can only learn from cases that are present. If people with incomes over $100,000 per year are a bad credit risk, and your training data includes nobody over $40,000 per year, you cannot expect it to make a correct decision when it encounters one of the previously-unseen cases. Extrapolation is dangerous with any model, but some types of neural network may make particularly poor predictions in such circumstances.

A network learns the easiest features it can. A classic (possibly apocryphal) illustration of this is a vision project designed to automatically recognize tanks. A network is trained on a hundred pictures including tanks, and a hundred not. It achieves a perfect 100% score. When tested on new data, it proves hopeless. The reason? The pictures of tanks are taken on dark, rainy days;

the pictures without on sunny days. The network learns to distinguish the (trivial matter of) differences in overall light intensity. To work, the network would need training cases including all weather and lighting conditions under which it is expected to operate - not to mention all types of terrain, angles of shot, distances...

Unbalanced data sets. Since a network minimizes an overall error, the proportion of types of data in the set is critical. A network trained on a data set with 900 good cases and 100 bad will bias its decision toward good cases, as this allows the algorithm to lower the overall error (which is much more heavily influenced by the good cases). If the representation of good and bad cases is different in the real population, the network's decisions may be wrong. A good example would be disease diagnosis. Perhaps 90% of patients routinely tested are clear of a disease. A network is trained on an available data set with a 90/10 split. It is then used in diagnosis on patients complaining of specific problems, where the likelihood of disease is 50/50. The network will react over-cautiously and fail to recognize disease in some unhealthy patients. In contrast, if trained on the "complainants" data, and then tested on "routine" data, the network may raise a high number of false positives. In such circumstances, the data set may need to be crafted to take account of the distribution of data (e.g., you could replicate the less numerous cases, or remove some of the numerous cases) (Bishop, 1995). Often, the best approach is to ensure even representation of different cases, then to interpret the network's decisions accordingly.

INSIGHTS INTO MLP TRAINING

More key insights into MLP behavior and training can be gained by considering the type of functions they model. Recall that the activation level of a unit is the weighted sum of the inputs, plus a threshold value. This implies that the activation level is actually a simple linear function of the inputs. The activation is then passed through a sigmoid (S-shaped) curve. The combination of the multi-dimensional linear function and the one-dimensional sigmoid function gives the characteristic sigmoid cliff response of a first hidden layer MLP unit (the figure below illustrates the shape plotted across two inputs. An MLP unit with more inputs has a higher-dimensional version of this functional shape). Altering the weights and thresholds alters this response surface. In particular, both the orientation of the surface, and the steepness of the sloped section, can be altered. A steep slope corresponds to large weight values: doubling all weight values gives the same orientation but a different slope.
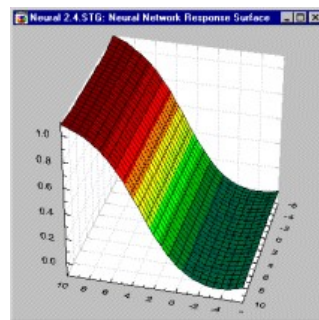

The next figure illustrates a typical response surface for a network with only one hidden layer, of two units, and a single output unit, on the classic XOR problem. Two separate sigmoid surfaces have been combined into a single U-shaped surface.

During network training, the weights and thresholds are first initialized to small, random values. This implies that the units' response surfaces are each aligned randomly with low slope: they are effectively uncommitted. As training progresses, the units' response surfaces are rotated and shifted into appropriate positions, and the magnitudes of the weights grow as they commit to modeling particular parts of the target response surface.

In a classification problem, an output unit's task is to output a strong signal if a case belongs to

its class, and a weak signal if it doesn't. In other words, it is attempting to model a function that has magnitude one for parts of the pattern-space that contain its cases, and magnitude zero for other parts.
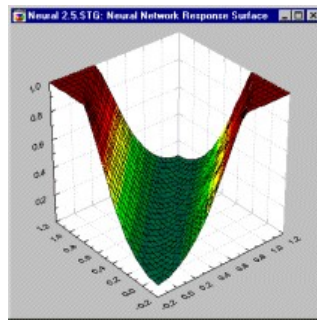

This is known as a *discriminant function* in pattern recognition problems. An ideal discriminant function could be said to have a plateau structure, where all points on the function are either at height zero or height one.

If there are no hidden units, then the output can only model a single sigmoid-cliff with areas to one side at low height and areas to the other high. There will always be a region in the middle (on the cliff) where the height is in-between, but as weight magnitudes are increased, this area shrinks.

A sigmoid-cliff like this is effectively a linear discriminant. Points to one side of the cliff are classified as belonging to the class, points to the other as not belonging to it. This implies that a network with no hidden layers can only classify linearly-separable problems (those where a line - or, more generally in higher dimensions, a hyperplane - can be drawn which separates the points in pattern space).

A network with a single hidden layer has a number of sigmoid-cliffs (one per hidden unit) represented in that hidden layer, and these are in turn combined into a plateau in the output layer. The plateau has a convex hull (i.e., there are no dents in it, and no holes inside it). Although the plateau is convex, it may extend to infinity in some directions (like an extended peninsular). Such a network is in practice capable of modeling adequately most real-world classification problems.
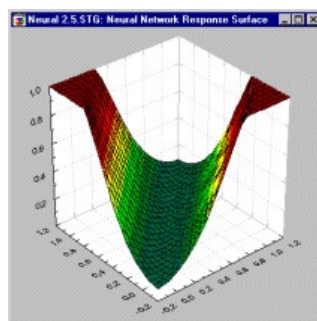
The figure above shows the plateau response surface developed by an MLP to solve the XOR problem: as can be seen, this neatly sections the space along a diagonal.

A key question in classification is how to interpret points on or near the cliff. The standard practice is to adopt some confidence levels (the accept and reject thresholds) that must be exceeded before the unit is deemed to have made a decision. For example, if accept/reject thresholds of 0.95/0.05 are used, an output unit with an output level in excess of 0.95 is deemed to be on, below 0.05 it is deemed to be off, and in between it is deemed to be undecided.

A more subtle (and perhaps more useful) interpretation is to treat the network outputs as probabilities. In this case, the network gives more information than simply a decision: it tells us

how sure (in a formal sense) it is of that decision. There are modifications to MLPs that allow the neural network outputs to be interpreted as probabilities, which means that the network effectively learns to model the probability density function of the class. However, the probabilistic interpretation is only valid under certain assumptions about the distribution of the data (specifically, that it is drawn from the family of exponential distributions; see Bishop, 1995). Ultimately, a classification decision must still be made, but a probabilistic interpretation allows a more formal concept of minimum cost decision making to be evolved.

### OTHER MLP TRAINING ALGORITHMS

Neural networks are highly nonlinear tools that are usually trained using iterative techniques. The most recommended techniques for training neural networks are the BFGS (Broyden-Fletcher-Goldfarb-Shanno) and Scaled Conjugate Gradient algorithms (see Bishop 1995). These methods perform significantly better than the more traditional algorithms such as Gradient Descent but they are, generally speaking, more memory intensive and computationally demanding. Nonetheless, these techniques may require a smaller number of iterations to train a neural network given their fast convergence rate and more intelligent search criterion.

Radial Basis Function Networks

We have seen in the last section how an MLP models the response function using the composition of sigmoid-cliff functions - for a classification problem, this corresponds to dividing the pattern space up using hyperplanes. The use of hyperplanes to divide up space is a natural approach - intuitively appealing, and based on the fundamental simplicity of lines.

An equally appealing and intuitive approach is to divide up space using circles or (more generally) hyperspheres. A hypersphere is characterized by its center and radius. More generally, just as an MLP unit responds (non-linearly) to the distance of points from the line of the sigmoid-cliff, in a radial basis function network (Broomhead and Lowe, 1988; Moody and Darkin, 1989; Haykin, 1994) units respond (non-linearly) to the distance of points from the center represented by the radial unit. The response surface of a single radial unit is therefore a Gaussian (bell-shaped) function, peaked at the center, and descending outwards. Just as the steepness of the MLP's sigmoid curves can be altered, so can the slope of the radial unit's Gaussian. See the next illustration below.
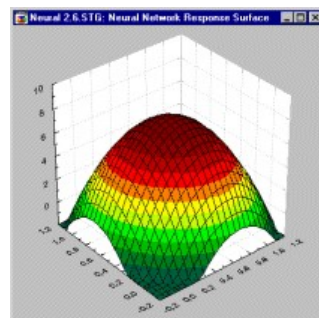
MLP units are defined by their weights and threshold, which together give the equation of the defining line, and the rate of fall-off of the function from that line. Before application of the sigmoid activation function, the activation level of the unit is determined using a weighted sum, which mathematically is the dot product of the input vector and the weight vector of the unit; these units are therefore referred to as dot product units. In contrast, a *radial unit* is defined by its center point and a radius. A point in N dimensional space is defined using N numbers, which exactly corresponds to the number of weights in a dot product unit, so the center of a radial unit is stored as weights. The radius (or deviation) value is stored as the threshold. It is worth

emphasizing that the weights and thresholds in a radial unit are actually entirely different to those in a dot product unit, and the terminology is dangerous if you don't remember this: Radial weights really form a point, and a radial threshold is really a deviation.

A *radial basis function* network (RBF), therefore, has a hidden layer of radial units, each actually modeling a Gaussian response surface. Since these functions are nonlinear, it is not actually necessary to have more than one hidden layer to model any shape of function: sufficient radial units will always be enough to model any function. The remaining question is how to combine the hidden radial unit outputs into the network outputs? It turns out to be quite sufficient to use a linear combination of these outputs (i.e., a weighted sum of the Gaussians) to model any nonlinear function. The standard RBF has an output layer containing dot product units with identity activation function (see Haykin, 1994; Bishop, 1995).

RBF networks have a number of advantages over MLPs. First, as previously stated, they can model any nonlinear function using a single hidden layer, which removes some design-decisions about numbers of layers. Second, the simple linear transformation in the output layer can be optimized fully using traditional linear modeling techniques, which are fast and do not suffer from problems such as local minima which plague MLP training techniques. RBF networks can therefore be trained extremely quickly (i.e., orders of magnitude faster than MLPs).

On the other hand, before linear optimization can be applied to the output layer of an RBF network, the number of radial units must be decided, and then their centers and deviations must be set. Although faster than MLP training, the algorithms to do this are equally prone to discover sub-optimal combinations. Other features that distinguish RBF performance from MLPs are due to the differing approaches to modeling space, with RBFs "clumpy" and MLPs "planey."

Other features which distinguish RBF performance from MLPs are due to the differing approaches to modeling space, with RBFs "clumpy" and MLPs "planey."

Experience indicates that the RBF's more eccentric response surface requires a *lot* more units to adequately model most functions. Of course, it is always possible to draw shapes that are most easily represented one way or the other, but the balance does not favor RBFs. Consequently, an RBF solution will tend to be slower to execute and more space consuming than the corresponding MLP (but it was much faster to train, which is sometimes more of a constraint).

The clumpy approach also implies that RBFs are not inclined to extrapolate beyond known data: the response drops off rapidly towards zero if data points far from the training data are used. Often the RBF output layer optimization will have set a bias level, hopefully more or less equal to the mean output level, so in fact the extrapolated output is the observed mean - a reasonable working assumption. In contrast, an MLP becomes more certain in its response when far-flung data is used. Whether this is an advantage or disadvantage depends largely on the application, but on the whole the MLP's uncritical extrapolation is regarded as a bad point: extrapolation far from training data is usually dangerous and unjustified.

RBFs are also more sensitive to the curse of dimensionality, and have greater difficulties if the number of input units is large: this problem is discussed further in a later section.

As mentioned earlier, training of RBFs takes place in distinct stages. First, the centers and deviations of the radial units must be set; then the linear output layer is optimized.

Once centers and deviations have been set, the output layer can be optimized using the standard linear optimization technique: the pseudo-inverse (singular value decomposition) algorithm (Haykin, 1994; Golub and Kahan, 1965).

However, RBFs as described above suffer similar problems to Multilayer Perceptrons if they are used for classification - the output of the network is a measure of distance from a decision

hyperplane, rather than a probabilistic confidence level. We may therefore choose to modify the RBF by including an output layer with logistic or softmax (normalized exponential) outputs, which is capable of probability estimation. We lose the advantage of fast linear optimization of the output layer; however, the non-linear output layer still has a relatively well-behaved error surface, and can be optimized quite quickly using a fast iterative algorithm such as conjugate gradient descent.

SOFM Networks

Self Organizing Feature Map (SOFM, or Kohonen) networks are used quite differently to the other networks. Whereas all the other networks are designed for supervised learning tasks, SOFM networks are designed primarily for *unsupervised learning* (see Kohonen, 1982; Haykin, 1994; Patterson, 1996; Fausett, 1994).

Whereas in supervised learning the training data set contains cases featuring input variables together with the associated outputs (and the network must infer a mapping from the inputs to the outputs), in unsupervised learning the training data set contains only input variables.

At first glance this may seem strange. Without outputs, what can the network learn? The answer is that the SOFM network attempts to learn the structure of the data.

One possible use is therefore in exploratory data analysis. The SOFM network can learn to recognize clusters of data, and can also relate similar classes to each other. The user can build up an understanding of the data, which is used to refine the network. As classes of data are recognized, they can be labeled, so that the network becomes capable of classification tasks. SOFM networks can also be used for classification when output classes are immediately available - the advantage in this case is their ability to highlight similarities between classes.

A second possible use is in novelty detection. SOFM networks can learn to recognize clusters in the training data, and respond to it. If new data, unlike previous cases, is encountered, the network fails to recognize it and this indicates novelty.

A SOFM network has only two layers: the input layer, and an output layer of radial units (also known as the *topological map* layer). The units in the topological map layer are laid out in space - typically in two dimensions (although *ST Neural Networks* also supports one-dimensional Kohonen networks).

SOFM networks are trained using an iterative algorithm. Starting with an initially-random set of radial centers, the algorithm gradually adjusts them to reflect the clustering of the training data.

The iterative training procedure also arranges the network so that units representing centers close together in the input space are also situated close together on the topological map. You can think of the network's topological layer as a crude two-dimensional grid, which must be folded and distorted into the N-dimensional input space, so as to preserve as far as possible the original structure. Clearly any attempt to represent an N-dimensional space in two dimensions will result in loss of detail; however, the technique can be worthwhile in allowing the user to visualize data which might otherwise be impossible to understand.

The basic iterative Kohonen algorithm simply runs through a number of epochs, on each epoch executing each training case and applying the following algorithm:

- Select the winning neuron (the one who's center is nearest to the input case);
- Adjust the winning neuron to be more like the input case (a weighted sum of the old neuron center and the training case).

The algorithm uses a time-decaying learning rate, which is used to perform the weighted sum and ensures that the alterations become more subtle as the epochs pass. This ensures that the centers settle down to a compromise representation of the cases which cause that neuron to win.

The topological ordering property is achieved by adding the concept of a neighborhood to the algorithm. The neighborhood is a set of neurons surrounding the winning neuron. The neighborhood, like the learning rate, decays over time, so that initially quite a large number of neurons belong to the neighborhood (perhaps almost the entire topological map); in the latter stages the neighborhood will be zero (i.e., consists solely of the winning neuron itself). In the Kohonen algorithm, the adjustment of neurons is actually applied not just to the winning neuron, but to all the members of the current neighborhood.

The effect of this neighborhood update is that initially quite large areas of the network are "dragged towards" training cases - and dragged quite substantially. The network develops a crude topological ordering, with similar cases activating clumps of neurons in the topological map. As epochs pass the learning rate and neighborhood both decrease, so that finer distinctions within areas of the map can be drawn, ultimately resulting in fine-tuning of individual neurons. Often, training is deliberately conducted in two distinct phases: a relatively short phase with high learning rates and neighborhood, and a long phase with low learning rate and zero or near-zero neighborhood.

Once the network has been trained to recognize structure in the data, it can be used as a visualization tool to examine the data. The *Win Frequencies* (counts of the number of times each neuron wins when training cases are executed) can be examined to see if distinct clusters have formed on the map. Individual cases are executed and the topological map observed, to see if some meaning can be assigned to the clusters (this usually involves referring back to the original application area, so that the relationship between clustered cases can be established). Once clusters are identified, neurons in the topological map are labeled to indicate their meaning (sometimes individual cases may be labeled, too). Once the topological map has been built up in this way, new cases can be submitted to the network. If the winning neuron has been labeled with a class name, the network can perform classification. If not, the network is regarded as undecided. Datasheet

SOFM networks are inspired by some known properties of the brain. The cerebral cortex is actually a large flat sheet (about 0.5m squared; it is folded up into the familiar convoluted shape only for convenience in fitting into the skull!) with known topological properties (for example, the area corresponding to the hand is next to the arm, and a distorted human frame can be topologically mapped out in two dimensions on its surface).

Classification in Neural Networks

In classification problems, the purpose of the network is to assign each case to one of a number of classes (or, more generally, to estimate the probability of membership of the case in each class). Nominal output variables are used to indicate a classification problem. The nominal values correspond to the various classes.

Nominal variables are normally represented in networks using one of two techniques, the first of which is only available for two-state variables; these techniques are: *two-state*, *one-of-N*. In two-state representation, a single node corresponds to the variable, and a value of 0.0 is interpreted as one state, and a value of 1.0 as the other. In *one-of-N* encoding, one unit is allocated for each state, with a particular state represented by 1.0 on that particular unit, and 0.0 on the others.

Input nominal variables are easily converted using the above methods, both during training and during execution. Target outputs for units corresponding to nominal variables are also easily determined during training. However, more effort is required to determine the output class assigned by a network during execution.

The output units each have continuous activation values between 0.0 and 1.0. In order to definitely assign a class from the outputs, the network must decide if the outputs are reasonably close to 0.0 and 1.0. If they are not, the class is regarded as undecided.

In an SOFM network, the winning node in the topological map (output) layer is the one with the lowest activation level (which measures the distance of the input case from the point stored by the unit). If the distance is small enough, the case is assigned to the class.

CLASSIFICATION STATISTICS

When selecting accept/reject thresholds, and assessing the classification ability of the network, the most important indicator is the *classification summary spreadsheet*. This shows how many cases were correctly classified, incorrectly classified, or unclassified. You can also use the confusion matrix spreadsheet to break down how many cases belonging to each class were assigned to another class. All these figures can be independently reported for the training, testing, and validation sets.

Regression Problems in Neural Networks

In regression problems, the objective is to estimate the value of a continuous output variable, given the known input variables. Regression problems are represented by data sets with non-nominal (standard numeric) output(s).

A particularly important issue in regression is output scaling, and extrapolation effects.

The most common neural network architectures have outputs in a limited range (e.g., (0,1) for the logistic activation function). This presents no difficulty for classification problems, where the desired output is in such a range. However, for regression problems there clearly is an issue to be resolved, and some of the consequences are quite subtle.

This subject is discussed below.

As a first pass, we can apply a scaling algorithm to ensure that the network's output will be in a sensible range. The simplest scaling function is minimax: this finds the minimum and maximum values of a variable in the training data, and performs a linear transformation (using a shift and a scale factor) to convert the values into the target range (typically [0.0,1.0]). If this is used on a continuous output variable, then we can guarantee that all training values will be converted into the range of possible outputs of the network, and so the network can be trained. We also know that the network's output will be constrained to lie within this range. This may or may not be regarded as a good thing, which brings us to the subject of extrapolation.
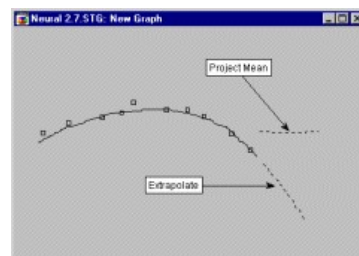
Consider the figure above. Here, we are trying to estimate the value of y from the value of x. A curve has to be fitted that passes through the available data points. We can probably easily agree

on the illustrated curve, which is approximately the right shape, and this will allow us to estimate y given inputs in the range represented by the solid line where we can interpolate.

However, what about a point well to the right of the data points? There are two possible approaches to estimating y for this point. First, we might decide to extrapolate: projecting the trend of the fitted curve onwards. Second, we might decide that we don't really have sufficient evidence to assign any value, and therefore assign the mean output value (which is probably the best estimate we have lacking any other evidence).

Let's assume that we are using an MLP. Using minimax as suggested above is highly restrictive. First, the curve is not extrapolated, however close to the training data we may be (if we are only a little bit outside the training data, extrapolation may well be justified). Second, it does not estimate the mean either - it actually saturates at either the minimum or maximum, depending on whether the estimated curve was rising or falling as it approached this region.

We can replace the logistic output activation function with a linear activation function, which simply passes on the activation level unchanged (N.B. only the activation functions in the output layer are changed; the hidden layers still use logistic or hyperbolic activation functions). The linear activation function does not saturate, and so can extrapolate further (the network will still saturate eventually as the hidden units saturate). A linear activation function in an MLP can cause some numerical difficulties for the *back propagation* algorithm, however, and if this is used a low learning rate (below 0.1) must be used. This approach may be appropriate if you want to extrapolate.

The above discussion focused on the performance of MLPs in regression, and particularly their behavior with respect to extrapolation. Networks using radial units (RBFs) perform quite differently, and need different treatment.

Radial networks are inherently incapable of extrapolation. As the input case gets further from the points stored in the radial units, so the activation of the radial units decays and (ultimately) the output of the network decays. An input case located far from the radial centers will generate a zero output from all hidden units. The tendency not to extrapolate can be regarded as good (depending on your problem-domain and viewpoint), but the tendency to decay to a zero output (at first sight) is not. If we decide to eschew extrapolation, then what we would like to see reported at highly novel input points is the mean. In fact, the RBF has a bias value on the output layer, and sets this to a convenient value, which hopefully approximates the sample mean. Then, the RBF will always output the mean if asked to extrapolate.

The performance of a regression network can be examined in a number of ways.

1. The output of the network for each case (or any new case you choose to test) can be submitted to the network. If part of the data set, the residual errors can also be generated.
2. Summary statistics can be generated. These include the mean and standard deviation of both the training data values and the prediction error. We would generally expect to see a prediction error mean extremely close to zero (it is, after all, possible to get a zero prediction error mean simply by estimating the mean training data value, without any recourse to the input variables or a neural network at all).
The regression statistics also include the Pearson-R correlation coefficient between the network's prediction and the observed values. In linear modeling, the Pearson-R correlation between the predictor variable and the predicted is often used to express correlation - if a linear model is fitted, this is identical to the correlation between the model's prediction and the observed values (or, to the negative of it). Thus, this gives you a convenient way to compare the neural network's accuracy with that of your linear models.

Time Series Prediction in Neural Networks

In time series problems, the objective is to predict the value of a variable that varies in time,

using previous values of that and/or other variables (see Bishop, 1995)

Typically, the predicted variable is continuous, so time series prediction is usually a specialized form of regression. However, without this restriction, time series can also perform prediction of nominal variables (i.e., classification).

It is also usual to predict the next value in a series from a fixed number of previous values (looking ahead a single time step). When the next value in a series is generated, further values can be estimated by feeding the newly-estimated value back into the network together with other previous values: time series projection. Obviously, the reliability of projection drops the more steps ahead we try to predict, and if a particular distance ahead is required, it is probably better to train a network specifically for that degree of lookahead.

Any type of network can be used for time series prediction (the network type must, however, be appropriate for regression or classification, depending on the problem type). The network can also have any number of input and output variables. However, most commonly there is a single variable that is both the input and (with the lookahead taken into account) the output. Configuring a network for time series usage alters the way that data is pre-processed (i.e., it is drawn from a number of sequential cases, rather than a single case), but the network is executed and trained just as for any other problem.

The time series training data set therefore typically has a single variable, and this has type input/output (i.e., it is used both for network input and network output).

The most difficult concept in time series handling is the interpretation of training, testing, validation, and ignored cases. For standard data sets, each case is independent, and these meanings are clear. However, with a time series network, each pattern of inputs and outputs is actually drawn from a number of cases, determined by the network's *Steps* and *Lookahead* parameters. There are two consequences of this:

The input pattern's type is taken from the type of the output case. For example, in a data set, the first two cases are ignored and the third is test, with *Steps=2* and *Lookahead=1*, the first usable pattern has type Test, and draws its inputs from the first two cases, and its output from the third. Thus, the first two cases are used in the test set even though they are marked Ignore. Further, any given case may be used in three patterns, and these may be any of training, testing, and validation patterns. In some sense, data actually leaks between training, testing, and validation sets. To isolate the three sets entirely, contiguous blocks of train, verify, or test cases would need to be constructed, separated by the appropriate number of ignore cases.

The first few cases can only be used as inputs for patterns. When selecting cases for time series use, the case number selected is always the output case. The first few clearly cannot be selected (as this would require further cases before the beginning of the data set), and are not available.

Variable Selection and Dimensionality Reduction

The preceding sections on network design and training have all assumed that the input and output layers are fixed; that is, that we know what variables will be input to the network, and what output is expected. The latter is always (at least, for supervised learning problems) known. However, the selection of inputs is far more difficult (see Bishop, 1995). Often, we do not know which of a set of candidate input variables are actually useful, and the selection of a good set of inputs is complicated by a number of important considerations:

Curse of dimensionality. Each additional input unit in a network adds another dimension to the space in which the data cases reside. We are attempting to fit a response surface to this data. Thought of in this way, there must be sufficient data points to populate an *N* dimensional space

sufficiently densely to be able to see the structure. The number of points needed to do this properly grows very rapidly with the dimensionality (roughly, in proportion to *2N* for most modelling techniques). Most forms of neural network (in particular, MLPs) actually suffer less from the curse of dimensionality than some other methods, as they can concentrate on a lower-dimensional section of the high-dimensional space (for example, by setting the outgoing weights from a particular input to zero, an MLP can entirely ignore that input). Nevertheless, the curse of dimensionality is still a problem, and the performance of a network can certainly be improved by eliminating unnecessary input variables. Indeed, even input variables that carry a small amount of information may sometimes be better eliminated if this reduces the curse of dimensionality.

Inter-dependency of variables. It would be extremely useful if each candidate input variable could be independently assessed for usefulness, so that the most useful ones could be extracted. Unfortunately, it is seldom possible to do this, and two or more interdependent variables may together carry significant information that a subset would not. A classic example is the two-spirals problem, where two classes of data are laid out in an interlocking spiral pattern in two dimensions. Either variable alone carries no useful information (the two classes appear wholly intermixed), but with the two variables together the two classes can be perfectly distinguished. Thus, variables cannot, in general, be independently selected.

Redundancy of variables. Often a number of variables can carry to some extent or other the same information. For example, the height and weight of people might in many circumstances carry similar information, as these two variables are correlated. It may be sufficient to use as inputs some subset of the correlated variables, and the choice of subset may be arbitrary. The superiority of a subset of correlated variables over the full set is a consequence of the curse of dimensionality.

Selection of input variables is therefore a critical part of neural network design. You can use a combination of your own expert knowledge of the problem domain, and standard statistical tests to make some selection of variables before starting to use Neural Networks. Once you begin using Neural Networks, various combinations of inputs can be tried. You can experimentally add and remove various combinations, building new networks for each. You can also conduct Sensitivity Analysis, which rates the importance of variable with respect to a particular model.

Another approach to dealing with dimensionality problems, which may be an alternative or a complement to variable selection, is *dimensionality reduction*. In dimensionality reduction, the original set of variables is processed to produce a new and smaller set of variables that contains (we hope) as much information as possible from the original set. As an example, consider a data set where all the points lie on a plane in a three dimensional space. The *intrinsic dimensionality* of the data is said to be two (as all the information actually resides in a two-dimensional sub-space). If this plane can be discovered, the neural network can be presented with a lower dimensionality input, and stands a better chance of working correctly.

Ensembles and Resampling

We have already discussed the problem of over-learning, which can compromise the ability of neural networks to generalize successfully to new data. An important approach to improve performance is to form ensembles of neural networks. The member networks' predictions are averaged (or combined by voting) to form the ensemble's prediction. Frequently, ensemble formation is combined with resampling of the data set. This approach can significantly improve generalization performance. Resampling can also be useful for improved estimation of network generalization performance.

To explain why resampling and ensembles are so useful, it is helpful to formulate the neural

network training process in statistical terms (Bishop, 1995). We regard the problem as that of estimating an unknown nonlinear function, which has additive noise, on the basis of a limited data set of examples, D. There are several sources of error in our neural network's predictions. First, and unavoidably, even a "perfect" network that exactly modeled the underlying function would make errors due to the noise. However, there is also error due to the fact that we need to fit the neural network model using the finite sample data set, D. This remaining error can be split into two components, the model bias and variance. The bias is the average error that a particular model training procedure will make across different particular data sets (drawn from the unknown function's distribution). The variance reflects the sensitivity of the modeling procedure to a particular choice of data set.

We can trade off bias versus variance. At one extreme, we can arbitrarily select a function that entirely ignores the data. This has zero variance, but presumably high bias, since we have not actually taken into account the known aspects of the problem at all. At the opposite extreme, we can choose a highly complex function that can fit every point in a particular data set, and thus has zero bias, but high variance as this complex function changes shape radically to reflect the exact points in a given data set. The high bias, low variance solutions can have low complexity (e.g., linear models), whereas the low bias, high variance solutions have high complexity. In neural networks, the low complexity models have smaller numbers of units.

How does this relate to ensembles and resampling? We necessarily divide the data set into subsets for training, testing, and validation. Intuitively, this is a shame, as not all the data gets used for training. If we resample, using a different split of data each time, we can build multiple neural networks, and all the data gets used for training at least some of them. If we then form the networks into an ensemble, and average the predictions, an extremely useful result occurs. Averaging across the models reduces the variance, without increasing the bias. Arguably, we can afford to build higher bias models than we would otherwise tolerate (i.e., higher complexity models), on the basis that ensemble averaging can then mitigate the resulting variance.

The generalization performance of an ensemble can be better than that of the best member network, although this does depend on how good the other networks in the ensemble are. Unfortunately, it is not possible to show whether this is actually the case for a given ensemble. However, there are some reassuring pieces of theory to back up the use of ensembles.

First, it can be shown (Bishop, 1995) that, on the assumption that the ensemble members' errors have zero mean and are uncorrelated, the ensemble reduces the error by a factor of N, where N is the number of members. In practice, of course, these errors are not uncorrelated. An important corollary is that an ensemble is more effective when the members are less correlated, and we might intuitively expect that to be the case if diverse network types and structures are used.

Second, and perhaps more significantly, it can be shown that the expected error of the ensemble is at least as good as the average expected error of the members, and usually better. Typically, some useful reduction in error does occur. There is of course a cost in processing speed, but for many applications this is not particularly problematic.

There are a number of approaches to resampling available.

The simplest approach is random (monte carlo) resampling, where the training, testing, and validation sets are simply drawn at random from the data set, keeping the sizes of the subsets constant. Alternatively, you CAN sometimes resample the training and test set, but keep the validation set the same, to support a simple direct comparison of results.

The second approach is bootstrap sampling. In the bootstrap, a new training set is formed by sampling with replacement from the available data set. In sampling with replacement, cases are

drawn at random from the data set, with equal probability, and any one case may be selected any number of times. Typically the bootstrap set has the same number of cases as the data set, although this is not a necessity. Due to the sampling process, it is likely that some of the original cases will not be selected, and these can be used to form a test set, whereas other cases will have been duplicated.

The bootstrap procedure replicates, insofar as is possible with limited data, the idea of drawing multiple data sets from the original distribution. Once again, the effect can be to generate a number of models with low bias, and to average out the variance. Ensembles can also be beneficial at averaging out bias. If we include different network types and configurations in an ensemble, it may be that different networks make systematic errors in different parts of the input space. Averaging these differently configured networks may iron out some of this bias.

Recommended Textbooks

Bishop, C. (1995). *Neural Networks for Pattern Recognition*. Oxford: University Press. Extremely well-written, up-to-date. Requires a good mathematical background, but rewards careful reading, putting neural networks firmly into a statistical context.

Carling, A. (1992). *Introducing Neural Networks*. Wilmslow, UK: Sigma Press. A relatively gentle introduction. Starting to show its age a little, but still a good starting point.

Fausett, L. (1994). *Fundamentals of Neural Networks*. New York: Prentice Hall. A well-written book, with very detailed worked examples to explain how the algorithms function.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Macmillan Publishing. A comprehensive book, with an engineering perspective. Requires a good mathematical background, and contains a great deal of background theory.

Patterson, D. (1996). *Artificial Neural Networks*. Singapore: Prentice Hall. Good wide-ranging coverage of topics, although less detailed than some other books.

Ripley, B.D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press. A very good advanced discussion of neural networks, firmly putting them in the wider context of statistical modeling.

Related link: StatSoft provides Data Mining and Predictive Analytics software and services. If you work for a business or government, you can request a STATISTICA Data Miner software trial. StatSoft Sales will contact you and discuss the options.