Chapter 5
Learning More About Neuralyst

Neural networks are really very simple in concept. However, like their biological counterparts, this simplicity is the foundation for highly complex behavior and sophisticated capabilities.

In chapter 4 the basic capabilities and operation of Neuralyst were discussed. This chapter focuses on several additional facets of Neuralyst and neural network behavior. The examples in this chapter illustrate these points as well as demonstrating a broad, though by no means complete, range of possible applications. At the start of each section, load the example indicated and explore it while reading the discussion. But don't stop there, experiment and see what happens!

The first six examples are based on idealized scenarios (much like `EXPLODE.XLS` `{Explode}`) with fairly simple rules of behavior. This has been done to allow the principles of behavior of neural networks to be demonstrated with fairly small data sets. In general, real data sets will be "noisier". That is, they will not have values that conform perfectly to a hidden model; instead the values will tend to vary around some "true" value with such variations being small to large depending on the circumstances. In order for neural networks to perceive the structure that may exist underlying such variations, more data must generally be presented and more time be spent in training.

In the next two examples, we will depart from the idealized situations and move on to real world data. Both of these will deal with investment analysis. The first is based on fundamental analysis, that is the forecasting of a stock or commodity's future price movements from data relating to a company's revenues, earnings, debt, equity, rates of returns, dividends, and so on. The second is based on technical analysis, that is the prediction of a stock or commodity's future price movements from past price movements. Fundamental analysis is generally considered a long-term approach, with forecasts ranging from many months to a few years. Technical analysis is generally considered a short-term approach, with predictions ranging from a few minutes to many weeks.

Finally, the last example provides a demonstration of Neuralyst's two-dimensional analysis and pattern matching capabilities through the recognition of patterns and shapes. Neuralyst's multi-dimensional analysis capabilities can be applied to a variety of applications, including: image processing, character recognition, and so on.

At the conclusion of the discussion and examples, you will have a much better understanding of the capabilities and limitations of neural networks and how to go about preparing a problem for Neuralyst to analyze.

5.1 Parity Generator — PARITY.XLS {Parity}

PARITY.XLS {Parity} contains a slightly more sophisticated example of a computer logic operation than shown in the first Neuralyst example, LOGIC.XLS {Logic}. Like that example, PARITY.XLS {Parity} works with the binary representation, 0's and 1's, of computer data and is a key function in computer operations.

PARITY.XLS {Parity} demonstrates the operation of *parity* generation for computer data. You may be aware that many computers, including PC's and Macintosh's, use the parity check operation to verify data in computer memory (you may have seen the message "PARITY CHECK" followed by a hung computer when the check fails on a PC).

Remember that computer data is stored in groups of eight bits, known as a *byte*. For each byte, the computer's parity circuits count the number of 1's present in the byte. If the count is odd, then there is odd parity; if the count is even, then there is even parity. When the byte is written, the parity is saved in a ninth bit, known as the parity bit. When the byte is read, the parity of the byte is checked against the previously saved parity bit. If there has been no change while the data resided in memory, then the parity as read will be the same as the parity when written. If there is a difference, then one of the bits must have changed (a 0 changing to 1 or a 1 changing to 0 will change the number of 1's and thereby disturb the parity) and the data has been corrupted. When this occurs, the computer stops since it is safer to stop operations than to try and continue with bad data that may result in additional problems.

In the PARITY.XLS {Parity} example, there are only four bits (sometimes known as a *nibble*) of input, rather than the eight bits present in a full byte. These four bits can represent any number from 0 to 15 and those are the values listed in the example. (If all eight bits had been used, the example would range from 0 to 255 — too many data lines for a simple demonstration.)

There are two Target columns, indicating the parity of the nibble, even or odd (zero 1's being counted as even). There are also two Output columns reserved for the neural network results. To run this example:

1. **Init Working Area** — starting at **N1**
2. **Set Rows** — **6** through **21**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns** — **C**, **D**, **E**, and **F**
4. **Add Target Columns** — **H** and **I**
5. **Add Output Columns** — **K** and **L**
6. **Set Network Size** — 3 Layers, 8 Hidden Neurons
7. **Set Network Parameters** — Training Tolerance to 0.2, Epochs per Update to 20

[If the steps described in this "short" form are not clear to you, please review Chapter 4, which goes through the entire process of configuring a network in great detail.]

With this configuration, you can start training. While this problem is training, pay particular attention to the RMS Error value. Normally, successful training is indicated by a steady decrease in the RMS Error value. If nothing seems to be happening after a while, try stopping the training and use the **Plot Training Error** command to look at the training progress. Resume training and look again after a while. You may notice that there sometimes periods when the error value doesn't seem to make much progress (it may even lose ground for a bit) and then there are other periods when the error value is reduced at a steady rate or even jumps downward. After some time Neuralyst will stop and the Output columns will match the Target columns; Neuralyst has learned to generate the parity of any four bit value!

As it turns out, the data in this problem has a characteristic that is particularly hard for neural networks. That is, very similar inputs lead to very different outputs. For every value in this example, there is another value that is different in only one input bit, yet has the opposite output value! Despite this, the neural network was able to learn the data. Still, this kind of characteristic in the input data can lead to some long training sessions if the problem data, training parameters, or network size are poorly set.

Even worse than data that is structured like this is *contradictory* data. That is data that has two or more input sets that match while they have very different outputs; for example, having the binary representation of 2 be even parity and later on having another instance where the binary representation of 2 is now odd parity. Such contradictions must be removed as the neural network generally cannot resolve these unless the error tolerance is set so loosely that the outputs are often useless.

Now, let's go back to the behavior we mentioned. Those periods, when error reduction seemed to make little progress, are known as *learning plateaus*. When this phenomenon occurs, it is generally believed that the neural network is undergoing a generalization process and developing internal representations of relevant characteristics of the data. In fact when these plateaus occur, the neural network is probably learning the most, even though the error value is changing the least!

Conversely, when the error value is making rapid progress in reduction, this generally means that the neurons have already sorted themselves out and the corrections are being applied with maximum effect to each neuron.

Sometimes it takes multiple plateaus, wherein new or additional distinctions or generalizations are made each time, followed by another phase of rapid error reduction, before the neural network is able to complete its learning.

Watch for this kind of behavior. This will help you in understanding what is happening with the neural network and may give an indication of whether the network has been properly sized for the problem.

To see what happens with marginally sized networks, rerun the problem by giving the **Set Network Size** command again. This time, set 3 layers and 4 hidden neurons. (You will be warned that this will cause any learning done so far to be forgotten; click on **OK**.) Then start training. What happens? Repeat this a few times. You will find that sometimes the neural network trains properly and other times it seems to reach a permanent plateau at some point, with Neuralyst continuing to run since it is not able to achieve the required error tolerance. There exists a solution for 4 neurons, but the neural network is not always able to find the solution!

The neural network training process can be thought of as an exploration of the weight space, all the different possible combinations of weight values, until a weight set is found that produces the desired targets. For this particular problem, with this particular neural network configuration, there exists *local minima* in the weight space. These are points in the weight space that “trap” the neural network; the backpropagation algorithm not being able to move out of that region to find the correct weight set. When local minima exist, one solution is to try increasing the number of neurons until the neural network is able to train consistently.

Another experiment to try is to change one or more lines of data so that contradictory cases are presented. (Anytime you change input or target data values, you must give the **Reload Network** command so that Neuralyst is aware that there have been changes and that it must pick them up.) Also, try different settings of the Training Parameters and Network Size on this problem, with and without contradictory data. Observe the learning behavior in each case.

Important Points:

> Neural networks have a difficult time learning when inputs having small distinctions between them require outputs with large distinctions between them.
>
> Neural networks cannot learn properly from contradictory data.
>
> Neural networks often experience learning plateaus; these are probably phases of neural network development during which distinctions and generalizations are made.
>
> Neural networks must have sufficient network capacity (size) to learn.

5.2 Paper-Rock-Scissors Game — PAPER.XLS {Paper Game}

PAPER.XLS {Paper Game} contains an example that is actually structurally very similar to PARITY.XLS {Parity}. Like that example, PAPER.XLS {Paper Game} works with the representations <PAPER, ROCK, SCISSORS> and <BONNIE, TIE, CHRIS> of possible input values and outcomes. As with PARITY.XLS {Parity}, the goal is to learn the rules of the game, and the rules define distinct and sometimes opposite outcomes for various changes in inputs.

Note that <PAPER, ROCK, SCISSORS> and <BONNIE, TIE, CHRIS> are ternary, that is three-valued, inputs and outputs. This example demonstrates two things, the symbolic capabilities of Neuralyst and the additional multi-valued capabilities of Neuralyst.

There are two Input columns that are set to the three possible choices of the two players Bonnie and Chris, that is Paper, Rock, or Scissors. There is a Target columns indicating who wins, or if it was a tie. There are also a matching Output columns reserved for the neural network results. To run this example:

1. **Init Working Area** — starting at **J1**
2. **Set Rows** — **6** through **15**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns** — **A**, and **B**

4. **Add Target Columns — D**
5. **Add Output Columns — F**
6. **Set Mode Flag Column — H**
7. **Set Mode Rows — 15** Set Symbol Row
8. **Set Network Size** — 3 Layers, 8 Hidden Neurons
9. **Set Network Parameters** — Training Tolerance to 0.2, Epochs per Update to 20

With this configuration, you can start training. After some time Neuralyst will stop and the Output column will match the Target column; Neuralyst has learned the rules of the Paper-Rock-Scissors game!

The comments and discussion for the PARITY.XLS {Parity} example are also relevant here and you can make many of the same experiments to learn more about how neural networks behave with different kinds of data.

Important Points:

Using symbolic representations provides a more natural way to express certain types of problems.

Neural networks can deal with a variety of multi-valued inputs and outputs.

5.3 Sine Wave — SINE.XLS {Sine}

SINE.XLS {Sine} contains an example of how Neuralyst can match and predict values for a complex mathematical function with just a few data points. In contrast to the computer logic operation that was shown in the PARITY.XLS {Parity} example, SINE.XLS {Sine} uses continuous real numbers rather than the binary representation, 0's and 1's, of computer data.

SINE.XLS {Sine} demonstrates the operation of *interpolation* on mathematical data. Many complex operations can be approximated by a mathematical function; for a given input value, there is a corresponding output value. For real world behavior, it is common to have a few sampled or measured values of the function, but not a complete description or every value of the function. From just a few samples, Neuralyst can often interpolate the values of the function that were not previously known.

The sine function is a familiar mathematical function from high-school trigonometry. It is an important function because it is used in every area of science and engineering. It is an interesting function because it is known as a *transcendental* function. Transcendental functions are harder to describe or generate than most functions that are familiar from high-school algebra. In particular, the values of the sine function are normally derived by computing an infinite series of terms. The input to a sine function is any real value, though in the example training is limited to 0 to 6.28, or 2π, and the output is any real value from -1 to 1.

In the `SINE.XLS` `{Sine}` example, there is only one Input and one Target column. There is also a corresponding Output column to match the Target column. To run this example:

1. **Init Working Area** — starting at **L1**
2. **Set Rows** — **5** through **140**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns** — **A**
4. **Add Target Columns** — **B**
5. **Add Output Columns** — **C**
6. **Set Mode Flag Column** — **D**
7. **Set Network Size** — 3 Layers, 3 Hidden Neurons
8. **Set Network Parameters** — Momentum to 0,
   Training Tolerance to 0.04,
   Epochs per Update to 50

With this configuration, you can start training. After a few minutes Neuralyst will be done training. In the example, some well-spaced samples for a single cycle of a sine wave are selected and used for training. The remaining points are used for testing and plotted against an exact sine wave for comparison. After training is complete do a Run/Predict to fill in the previously unknown points. Look at the resulting comparison chart. Neuralyst has generalized the shape of a sine wave from just a few sample points!

The interpolation is off by only a few percent at the worst points and at many points is almost exact. The function can be made more exact with more training time, more neurons or more data points used for training. More training time allows the neural network more time to adjust its weights to a better solution. More neurons allows the neural network more capacity to develop a model of the sine wave. More data points gives the neural network more information to constrain the approximation of the sine wave at those points that are changing rapidly and are far away from an input training case. Try experimenting with which of the three variations is most successful in generating a more exact interpolation. Also observe that there is a limit to how exact the neural network can be.

In addition to the above experiments, Neuralyst allows you to control two parameters, Calculation Method and Scaling Margin in the **Set Enhanced Parameters** dialog box, which can also affect speed of training, precision and accuracy. Try training with Calculation Method set to Floating Point versus the default Fixed Point method. Also try adjusting the Scaling Margin from 10% to 50%. While adjusting those two parameters, try tightening Training Tolerance to 0.03, 0.02 ore even 0.01 (that is, 3%, 2% or even 1%).

Important Points:

> Neural networks can approximate and interpolate continuously valued functions with relatively few training points.
>
> Despite the capabilities available with relatively few training cases, more training cases will generally provide better training.
>
> The choice of Calculation Method can affect the training of certain types of problems.

The setting of Scaling Margin can also affect the training of certain types of problems.

5.4 Criminal Mugbook — MUGBOOK.XLS {Mug Book}

So far there have been three examples, LOGIC.XLS {Logic}, PARITY.XLS {Parity} and PAPER.XLS {Paper Game}, which have demonstrated a neural network's capabilities to learn and reproduce rules from examples of those rules and two examples, EXPLODE.XLS {EXPLODE} and SINE.XLS {SINE} which have demonstrated a neural network's capabilities to generalize and predict from known facts. MUGBOOK.XLS {Mug Book} will demonstrate an example of how neural networks can also be used for *pattern matching* or as an *associative memory*.

You are probably familiar with the concept of a mugbook, a book of photos used by the police to help witnesses match the physical characteristics of known criminals against the features of a suspect the witness has seen. In some cases, there are problems in the identification process since the witness is not completely sure of the match due to the uncertainty of their memory, poor visibility at the time of the crime or changes in outward characteristics, for example, weight gain, shorter hair length or deliberate disguise.

MUGBOOK.XLS {Mug Book} is a simplified example of a mugbook, based on four physical characteristics: sex, age, coloring, and weight of eight known criminals in Midtown, U.S.A.. Most of these characteristics have been converted to a symbolic value, using the translation shown under each column of the worksheet. For example, age has been broken into decades, coloring has been segmented into three groupings: light, medium, and dark, and so on. Each of the eight criminals has also been assigned an ID number from 1 to 8.

The Target and Output columns are set up as eight separate indicators, each one representing a different ID. A 1 under an ID indicates that the criminal with that ID is completely identified, a 0 under an ID indicates that criminal is completely rejected. For a solid ID, all indicators should be 0 except for one that contains 1. There are two reasons why the outputs have been organized in this fashion. Let's discuss these for a moment.

First, while neural networks can make fine distinctions, there is a limit to the number of distinctions, that is different output values, that a single neuron can meaningfully take on. This can result in self-deception if you are not careful. You can train the neural network to produce the actual output values to match the target values (within the Training Tolerance), no matter how fine the distinction, and so believe that the neural network has made these distinctions. But when the neural network is run, these distinctions will not be successfully reproduced.

There is no hard rule as to how many distinctions can be made successfully by an output, but numbers beyond 4 to 8 are generally difficult. In this case, with 8 ID's to match, we have established a separate output neuron for each ID.

The second reason anticipates the conclusion of the demonstration to a certain extent. Once the network has been trained on the known criminals, we will present the characteristics of an unknown to try and match against the known ones. The use of separate outputs for each ID allows the neural network to use the full output range to indicate the quality of the match for the known characteristics of each ID against the characteristics of the unknown person.

To clarify this some more, if an unknown had some of the characteristics of ID 3 and some of the characteristics of ID 5, the only way a single output could express this would be by presenting 4. You would have no way of distinguishing this output from an actual match with ID 4 or partial matches between two or more ID's that averaged 4. With separate outputs, the outputs for ID 3 and ID 5 could each present a fractional value, indicating a partial match, without any confusion.

To run this example:

1. **Init Working Area** — starting at **AC1**
2. **Set Rows — 9** through **18**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns — D**, **E**, **F**, and **G**

4. **Add Target Columns — I** through **P**
5. **Add Output Columns** — **R** through **Y**
6. **Set Mode Flag Column — AA**
7. **Set Mode Rows — 18**, Set Symbol Row
8. **Set Network Size** — 3 Layers, 6 Hidden Neurons
9. **Set Network Parameters** — Training Tolerance to 0.2, Epochs per Update to 20

With this configuration, you can start training. After a short time Neuralyst will be done. Neuralyst has learned the distinguishing characteristics of the eight known criminals! Any suspect with exactly the same characteristics as one of these known criminals will immediately produce a match. This capability is similar to the rule reproduction capability already demonstrated before.

However, this is somewhat more powerful than may be obvious from this simplified example. The reason is that this capability can be extended to hundreds of characteristics and thousands of criminals (or any other search objects). This is the same function performed by conventional computer databases. However, computer designers know that searching and matching in large databases are among the most time consuming database operations. On the other hand, a neural network trained to the same data as a large database could "retrieve" a match with just one processing operation!

A more sophisticated capability than this will soon be apparent. Select **Run/Predict with Network** from the **Neural** menu to run the network. The characteristics of Mr. X are processed and the eight outputs in that row now have fractional values in them. These fractional values represent the neural network's assessment of how closely Mr. X matches characteristics of the known criminals. The neural network is able to generate indications for the closest matches even though it didn't find an exact match!

The strongest outputs are likely to be for ID 5 and ID 7, with other outputs perhaps showing a response. While there is some information in the relative values of the matches, these values should not be taken at first inspection as exact measures of closeness or probability. There are three reasons for this.

First, remember that the Training Tolerance was set to 0.2 or 20% of the output range, thus we can't expect predictions to 5% when we trained to tolerances of 20%. However, there is a danger to training too strictly. It is possible that a neural network tries so hard to match the exact values in training that it loses its generalizations. This is called *overtraining*. This is particularly likely to occur when there are too many hidden layer neurons, too few training samples and too much training time. In this case, what happens is that the neural network has so much capacity in relation to the data it must learn, that it can afford to match the outputs rather than to generalize. In essence, it is easier for the neural network to build an internal "crib sheet" rather than understand the structure of the data!

Second, we don't know exactly what characteristics the neural network has determined are relevant (this is particularly important with small sample sets, as in these examples, where there will be fewer or no samples to contradict bad generalizations) and there is often no way to find out without experimenting with the neural network. Theoretically it should be possible to understand the model developed by the neural network through a detailed examination and understanding of the weights, but in practice these values and relationships are often too complex for this to be attempted.

Third, for complex systems, the weights that a neural network uses to begin training (which are randomly assigned with each new configuration) can determine which one of a few (or many) possible solutions is actually found. This is why rerunning some of these problems with a new set of weights may result in slightly different solutions. The fact that the results are sometimes slightly different doesn't mean that the neural net is giving incorrect answers. Instead, it means that the data presented to the neural network admits of more than one solution.

Having considered these limitations, in the context of this example and with our current understanding of this neural network's behavior, it is best to say that the neural network considers ID 5 and ID 7 to be strong candidates, while the other ID's with weaker outputs are weaker candidates, without placing too much emphasis on exactly how much stronger or weaker these candidates are with respect to each other.

However, you can experiment with this neural network's behavior. When you have learned enough, perhaps you could say more. Try decreasing the Training Tolerance by steps of 0.05 from 0.4 to 0.1, training on the known criminals and running on the unknown after each change. You may want to try using the User Set Randomization option with **Reset Weights** for these experiments in order to start from a standard set of initial weight values (see Section 8.2.9). What happens to the values of the outputs? Try increasing the number of hidden layer neurons in the neural network and retraining. What happens when the unknown is matched now? Try adding more criminals with characteristics spaced evenly from each other and those already in the training set. Does the neural network do a better job of finding matches?

Important Points:

Neural network outputs should not be designed to make many fine distinctions.

Several separated neural network outputs can convey more information than a few combined outputs.

Setting the Training Tolerance more tightly than is necessary may interfere with generalization within the neural network.

Too much network capacity (size) and excessive training time may let the neural network "crib" rather than learn.

Neural networks require comprehensive, well-sampled training data in order to develop good generalizations

5.5 Credit Rater — EZCREDIT.XLS {EZ Credit}

`EZCREDIT.XLS {EZ Credit}` provides another demonstration of neural network pattern matching. `EZCREDIT.XLS {EZ Credit}` is similar in concept and structure to `MUGBOOK.XLS {Mug Book}`, but it provides an example of how the input types may also be categorized by indicators using binary values, in a similar fashion to the outputs in MUGBOOK.XLS {Mug Book}.

In this demonstration the Credit Approval Manager for EasyCredit Corporation has set up a database containing individuals distinguished by four characteristics: Income, Credit Experience, current Debt Burden, and prior Bankruptcy status. These characteristics are then matched to the actual credit history of the individuals EasyCredit has compiled from working records. EasyCredit expects that once the neural network is taught on its database of current clients, it will be able to use the neural network to rate new applicants.

Since the information contained in the credit records is varied, some numerical, some categorical, and some yes/no types, the manager has chosen to break each input type into one or more classifications or subdivisions that she feels are meaningful without being too fine. For each input type, the valid classification will be indicated by a 1, while the other classifications will be indicated by a 0. For example, for Income, she has chosen three classes: 0-30, 30-60, and 60+. She knows that these income ranges tend to define breaks where people have moderate, good, and excellent, ability to repay loans, respectively.

It is also possible that more than one class may be valid. For example, in the case of Credit Experience, she has identified the three most meaningful classifications as those people who have no credit cards, those with a department store credit card or those with a major bank credit card. Since a person can have both store credit cards and bank credit cards, a 1 could be entered for each subdivision, if appropriate.

Each of the four major credit characteristics have been classified in this way and entered into the worksheet for 16 customers. For these customers, the credit risk, represented by the payment history actually experienced by the company, is listed as Low, Medium, or High. A test case, Joe Applicant, is shown in the last row.

To run this example:

1. **Init Working Area** — starting at **AA1**
2. **Set Rows — 8** through **24**, 1 Row/Pattern, 1 Row/Shift

3. **Add Input Columns — C**, **D**, **E**, **G**, **H**, **I**, **K**, **L**, **M**, and **O** (Note the omission of columns **F**, **J**, and **N**! Perform this operation as four separate Add Input Columns — C,D,E then G,H,I then K,L,M then O.)
4. **Add Target Columns — Q**, **R**, and **S**
5. **Add Output Columns — U**, **V**, and **W**
6. **Set Mode Flag Column — Y**
7. **Set Network Size** — 3 Layers, 4 Hidden Neurons
8. **Set Network Parameters** — Training Tolerance to 0.2,
   Testing Tolerance to 0.4,
   Epochs per Update to 10

With this configuration, use the **Train Network** command to begin training. Neuralyst will stop after a short time. At this point it has taken the credit records of EasyCredit's past customers and established from this database the characteristics that contribute to credit rating and whether each characteristic does so positively or negatively!

When it is done, use the **Run/Predict with Network** command to evaluate the prospects of Joe Applicant. You will find that Neuralyst predicts Joe will most likely be a Medium credit risk, though he has a few characteristics of a High credit risk. This matches the Medium credit risk rating given to him by our (hidden) scenario rules.

There is nothing preventing finer subdivisions. Income could be broken down into increasing increments of ten thousand. Bank credit cards could be expanded to separately indicate Visa, MasterCard, or American Express. It is quite possible that the neural network will be able to make finer judgments with this additional information. The disadvantage to much finer subdivisions is the cost of maintaining them when establishing or updating the database and the additional computation time for the neural network with more inputs to consider. Your experience and judgment should guide the process.

Important Points

Separated or categorized input values can be used to convey information to the neural network in a more efficient way.

Too many categories can be burdensome to maintain and cost additional computation time needlessly.

Your experience and judgment should guide the process to make the most meaningful distinctions.

5.6 Marketing Analyzer — FIZZY.XLS {Fizzy Cola}

The demonstrations discussed so far have shown how Neuralyst can be used for rule reproduction, generalization, prediction, pattern matching, and association. FIZZY.XLS {Fizzy Cola} will demonstrate one way in which neural networks can be used to analyze data.

In FIZZY.XLS {Fizzy Cola}, we meet the Vice President for Sales of Fizzy-Cola. The Fizzy V.P. has divided the National market into eight regions and assigned each region to a manager that reports to him. As a great believer in decentralized management, he has allowed each Regional Manager to allocate their advertising budget independently of the others. The Fizzy V.P. is also scrupulously fair as he has made sure that each region has an equivalent amount of advertising money to spend in proportion to their population base.

Advertising money can be spent in four basic ways: In-store promotions (for example, store displays, price discounting), direct mail (of coupons or other promotional offers to homes), print media (newspaper or magazine advertisements), and radio/TV (commercials). When he reviews the results for the current quarter, he discovers that each Regional Manager has developed a unique allocation of advertising dollars for these four primary categories. He also determines that the sales growth in each region has varied greatly.

Of course, it would be possible to encourage the other regions to duplicate the budget allocation developed by the Regional Manager with the best sales results, but the Fizzy V.P. would like to find out if an even more successful allocation can be developed using the information contained in the current quarterly report.

The Fizzy V.P. has entered the report data into a worksheet. The budget data has been listed by advertising category and region. Since the different regions are not all exactly the same size, he has eliminated population and other base factors by listing expenditures in dollars per 1000 capita instead of total dollars and sales as percentage growth rather than total dollars.

In addition to the standard data rows, there are five more rows. The first four rows of these will be used to "probe" the neural network, once Neuralyst has learned the relationships between the different advertising budgets and each region's sales performance. The probing is done by taking each category in turn, and setting it to the maximum value known for that category while setting the others to the minimum values known for those categories. In this case, there are four advertising categories, so there are four rows set up for probing. Click on the cells in the range **C14** to **F17** to see the Excel formulas used to generate the maximum or minimum values.

In each one of these cases, the probe will maximize one of the neural network's inputs, while minimizing all the others. In this way, we can try and quantify the response of a neural network to individual inputs.

The Fizzy V.P. will use the information garnered from this probing to develop a new budget allocation, which we will test for him by entering into the last row.

To run this example:

1. **Init Working Area** — starting at **N1**
2. **Set Rows — 6** through **17**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns — C**, **D**, **E**, and **F**
4. **Add Target Columns — H**
5. **Add Output Columns — J**
6. **Set Mode Flag Column — L**
7. **Set Network Size** — 3 Layers, 4 Hidden Neurons
8. **Set Network Parameters** — Epochs per Update to 10

Train the neural network with this configuration. Neuralyst will be active for a short time and then complete its training. At this point, Neuralyst has discovered the underlying relationships between Fizzy-Cola's advertising expenditures and sales performance!

Once the neural network is trained, run the neural network so that the probe rows will be evaluated. When that is complete, you will see the results of the probe in cells **J14** through **J17**. The greatest output occurs for Radio/TV, second is In-store, third is Print Media, and last is Direct Mail.

Test this result by placing the values 5, 1, 2, and 10 in the cells **C18** to **F18**, respectively, of the Test Budget row. **H18** has already been programmed with the formula used to model the sales performance in the other cases of this scenario. You will find that the resulting predicted sales growth of 26.75% is 0.5% higher than the best previous case, the Northwest region. The Fizzy V.P. has achieved his goal of bettering the prior sales performance using data analysis from Neuralyst.

The technique shown here can be very useful; however, you should always test the results for sensibility before using them as it is possible to go astray.

First, it is important to minimize the number of effects that are being analyzed, so that the effect of each factor can be seen more clearly. If several factors are changing at the same time, then it will be difficult to untangle the knot of inter-relationships. One way to do this is to use ratios and percentages rather than absolute numbers. Another is to hold parameters constant where possible. In the example here, the ratio, dollars per capita, was used as input, the sales growth, as a percentage, was used as the output and the total of advertising dollars for each population unit was constant.

If the total advertising dollars in proportion to the population base had not been constant, would that make the data impossible to analyze? No. It would mean more probe cases would be needed, in this case with varying totals so the response of the model to different total amounts could be measured. Try experimenting with this case.

Second, this case is simplified in that all the inputs contributed positively to the output result. In most cases, some of the inputs will contribute negatively, that is the more the input is increased the more the output is reduced. This case needs to be distinguished from the simpler case where the input has little effect on the output. Recognizing these distinctions is important to a correct analysis.

Additionally, there are times when two or more inputs may interact with each other. These cases cannot be detected with probe cases that only have one input set to the maximum. An example of this can be seen in the LOGIC.XLS {Logic} or PARITY.XLS {Parity} demonstrations. In either of those worksheets, the presence of one 1 on an input results in a 1 on the output, yet two 1's results in a 0 — not a 2. If you suspect that this may be occurring, then probe cases where two or more inputs are set to their maximum values can be used.

Finally, in more complex cases, it may be useful to use probe cases where one input varies by fixed increments, for example, 10% of the input range per case, while the other inputs are held constant, usually at the minimums. This will result in a response curve, which can be plotted with Excel's charting capabilities. Each input can be probed in this way, resulting in a family of response curves that can be studied to determine the characteristics of the internal model developed by the neural network.

Important Points

Methodical probing of the neural network can lead to a successful analysis of the input data and its underlying relationships.

The number of parameters being measured should be minimized to ease the difficulty of interpreting results.

Positive, negative and inter-related effects should all be considered and probe cases created to test for them if appropriate.

In some cases, generating response curves for each input may be useful in achieving a successful analysis.

5.7 Fundamental Stock Analysis – AMETEK.XLS {Ametek}

In AMETEK.XLS {Ametek}, fundamental stock data (real world!) for the last twenty years for a smaller (annual revenues about $800 Million) New York Stock Exchange listed stock, Ametek (ticker symbol AME), have been entered into the worksheet. This kind of data is readily available from a variety of sources. Two popular ones are the Standard & Poors stock data sheets and the Value Line Investment Survey stock reviews.

The data is primarily organized on a per share basis. These are: sales revenue per share (Sls/Sh), cash flow per share (CF/Sh), earnings per share (Ern/Sh), dividends per share (Div/Sh), capital spending per share (Cap$/Sh), book value per share (BV/Sh), average price to earnings ratio for the year (Avg P/E), relative price to earnings ratio for the year compared to the overall market (Rel P/E), dividend yield (Div %), and the average price per share for the year (Avg $/Sh). (If you do not understand the terms used here please consult a stock investment book to learn the significance of these and other fundamental measures.)

While we could apply Neuralyst to this data directly, it would not be the most effective way to present the data to the neural network. Remember that neural networks work better if they are not required to make many fine distinctions in the input values. While we don't know if the distinctions between the values in this example will be critical, it is obvious that the values for many of the inputs take a different value for each row. Thus we should assume that each of these could be important to a successful forecast.

In order to satisfy the need to present the full range of the data to the neural network while also resolving the need to minimize the number of distinct values, we can present the differences between values. Thus a 1 point change in an input value with a full range of, for example, 10 to 25 would only represent a 6% change presented in this way. However, a 1 point change might represent 50% of the maximum change from year to year when presented in the context of differences between values.

There is another problem. Not all 1 point changes are equal! For example, a 1 point change from a base of 10 is more significant than a 1 point change from a base of 25. The first represents a 10% change, while the second represents a 4% change. One way to resolve this discrepancy is to take the logarithms of the input values. Logarithms have the property that a given percentage change in an input value, regardless of the starting point of the input value, will always be represented by the same change in logarithmic value. Thus, a 50% increase, whether starting from 10 or from 25, would always be represented by a change of 0.41 in the natural logarithm (it doesn't matter whether common or natural logarithms are used as long as the usage is consistent).

However, taking differences or logarithms may not make sense if the relationship between instances is not structured in time or some other dependent fashion. For example, taking differences between instances in MUGBOOK.XLS {Mug Book} would not make sense. This is because there is no reason to expect any relationship or special order between the processing of one criminal and the next criminal.

In those problems where there is a structured relationship, such as time, between instances, examples, or cases, these two methods are individually applicable. They can also be combined by taking the differences of the logarithms of the input values.

In order to implement the techniques just described, the differences of the logarithms of each input value from the previous input value have been computed in a new area just below the original area. (Note that the differences of the logs of two values is the same as the log of the ratio of those values. Though we describe it as differences, the formulas programmed are expressed in the ratio form since only one log is computed rather than two in this form.) Since each row in this area requires two rows from the original area so it can be computed, the first year, 1974, can no longer be shown.

Once the new area has been set up, we need to establish the training targets. In this case, we take advantage of future knowledge. Basically, we are setting the neural network to find any relationships that may exist that can be correlated to, or used to forecast, what will happen in the succeeding year during the training process. When the training process has ended, the future knowledge will no longer be available - but by then the relationships that could forecast that future may have been uncovered.

The Buy training target is established by "peeking" ahead to the next year and checking if the stock price has risen by at least 20% from the current year. If it has, then that is deemed a positive movement and the stock should be purchased in the current year, indicated by a BUY in the Buy column. Click on the cell **N29** (or similar cell in column **N**) to see the formula used to generate this target.

The Sell training target is generated in a similar fashion, but in its case if the stock has not at least retained its current price, then that is deemed a negative movement and the stock should be sold in the current year, indicated by a SELL in the Sell column. Click on the cell **O29** (or similar cell in column **O**) to see the formula used to generate this target.

(Note that the last row has no targets for training or testing. Since we cannot really look into the "future", except in hindsight, the number of rows that we "peek" ahead determines the number of rows that must be left blank at the end of the Target columns.)

To run this example:

1. **Init Working Area** — starting at **V1**
2. **Set Rows** — **29** through **48**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns** — **C** through **L**
4. **Add Target Columns** — **N** and **O**
5. **Add Output Columns** — **Q** and **R**
6. **Set Mode Flag Column** — **T**
7. **Set Mode Rows** — **48**, Set Symbol Row
8. **Set Network Size** — 3 Layers, 6 Hidden Neurons
9. **Set Network Parameters** — Epochs per Update to 10

Train the neural network on the data with this configuration. After some time Neuralyst will stop training. Has Neuralyst uncovered relationships that can be used to forecast the price performance of Ametek stock? Try running the neural network to make a forecast for the most recent year, 1994. The forecast will likely be to Sell Ametek stock for 1994, given 1993's data. As of the publication date of this manual, that may or may not have been a good forecast. This is because the current price of Ametek is up 20% primarily due to a 20% stock repurchase that occurred in 1994. This is a reminder that a neural network cannot predict events for which no training or modeling has been done.

In fact, this forecast, while it is useful for this demonstration, should not be relied upon at this point, independent of the probable outcome of a single prediction. In this case, there are only 17 data sets available to train the neural network. For real-world data, particularly data that is as noted for its “noise” content as stock data is, much more training data should be presented before the forecasts of the neural network should be considered in any serious way.

This can be done by going backward and presenting data from more previous years than shown in this example. Unfortunately, this approach may be difficult to implement for at least two reasons. First, data much older than this is not as readily available. Second, economic, competitive and other structural conditions often change significantly over such a long duration. In the case of Ametek, for example, it was a much different company in the 1950’s and 1960’s than it has been in the 1970’s and 1980’s. In order for the neural network to identify the relationships between input factors while these underlying conditions are changing, even more data must be presented so enough cases representative of their effects can be seen by the neural network. After a certain point, this can become an impossible task. As an example, events such as the aforementioned stock repurchase are rare or unique events which have few precedents.

Another approach that is more likely to be successful is to present data from a large number of companies during the same time period. This would hold certain implicit factors constant, for example general economic climate, interest rates, credit availability, inflation, and so on, allowing the neural network to measure the factors that lead to relative differences in performance. Two specific variations on this approach would be: 1) to train the neural network on companies that are in the same industry or produce the same product, or 2) to train the neural network across the spectrum of companies that are structured in similar ways, for example conglomerates or highly-leveraged companies.

Important Points:

Taking differences between input values is a useful technique for improving the ability of the neural network to interpret the data.

Taking logarithms of input values is another useful technique for improving the ability of the neural network to interpret the data.

Combining the two techniques of differences and logarithms is also useful.

For either or both of these techniques to work there should be a structured relationship, such as time, between instances, examples or cases.

Training a neural network for forecasting usually requires some use of “future knowledge”.

Be sure there is enough training data and the neural network is tested thoroughly before you rely on its predictions.

Be sure the training data you use does not contain more underlying conditional variations than you want the neural network to consider.

5.8 Technical Stock Analysis — DJIA.XLS {DJIA}

In DJIA.XLS {DJIA} price data on a weekly basis for the Dow Jones Average of 30 Industrial stocks from the beginning of January 1993 through September 1994 have been entered into the worksheet. This data consists of the highest price for the week, the lowest price for the week, the closing price on the week and the total volume of stocks traded in the market. This kind of data, whether on a monthly, weekly, daily, hourly, or even minute-by-minute basis is the starting point of most technical analysis methods.

We will use another set of techniques, as distinguished from the differences-of-logs form used in AMETEK.XLS {Ametek}, to pre-process raw price data into more meaningful forms. These will include a differences of inputs over time and moving averages. (Some of the pre-processed columns in this example were actually generated by the accompanying package Trader's Macro Library. See Appendix G for a discussion of how to load and use this macro toolbox for technical investment analysis.)

In the first pre-processed input column, the Close of the current work is detrended by taking the difference from the previous week, this is labeled Delta Close. The second pre-processed input column is generated by taking the ratio of the current Close to a previous Close as a percentage, this is labeled ROC or Rate of Change. The third pre-processed input column is generated by taking the difference of the Close of the current week from the Close five weeks ago. This difference over long periods of time is called "Momentum" by technical investment analysts. Finally, the last pre-processed input column is filled with the difference between a five week Moving Average of the Close for each week and a three week Moving Average of the Close for each week. This difference of Moving Averages of different periods is known as a Moving Average Oscillator by technical investment analysts. (Notice that the ROC, Momentum and Moving Average cannot be computed until a number of weeks of data are available. This will result in the first five rows being skipped when we give the **Set Rows** command.)

In setting up the targets for training, we will make use of future knowledge as in AMETEK.XLS {Ametek}. Our Buy and Sell targets will be determined by the future value of the Close column. If the Close for the next week will be higher than the Close for this week, then the Buy target will be set for this week. If the Close for the next week will be lower than the Close for this week, then the Sell target will be set for this week.

(As in AMETEK.XLS {Ametek}, the last row has no targets for training or testing. Since we cannot look into the "future", except in hindsight, the number of rows that we "peek" ahead determines the number of rows that must be left blank at the end of the Target columns.)

We will use a new technique in this neural network example, that of *iterated data windows*. So far in these examples, we have only set the neural network to train on one row of the worksheet at a time. For time based problems, this corresponds to making predictions while looking at data from just one point in time. In fact, there is every reason to believe that the relationships between values at different points in time are also significant to successful prediction. To do this we will set the number of Rows per Pattern in the problem definition to be 5. This corresponds to looking at the most recent 5 weeks of price data in every prediction. With this setting, Neuralyst will present 5 weeks at a time to the neural network, stepping one week each time.

To run this example:

1. **Init Working Area — starting at U1**
2. **Set Rows — 11** through **97**, 5 Rows/Pattern, 1 Row/Shift
3. **Add Input Columns — H** through **K**
4. **Add Target Columns — M** and **N**
5. **Add Output Columns — P** and **Q**
6. **Set Mode Flag Column — S**
7. **Set Mode Rows — 97**, Set Symbol Row
8. **Set Network Size** — 3 Layers, 12 Hidden Neurons
9. **Set Network Parameters** — Training Tolerance to 0.2,
   Testing Tolerance to 0.4,
   Epochs per Update to 10

Train the neural network with this configuration. After a short time Neuralyst will stop training. Has Neuralyst learned to predict the price performance of the Dow Jones Industrial Average from price data alone? Try running the neural network on the remaining data and compare the results to the targets. How did it do?

Most likely the matches are good, perhaps 60-70%, but not as high as we are used to from prior examples. Technical analysis of stock (or commodity) prices is a difficult problem and generally any predictive results with an accuracy better than 50% could provide an important edge in a trading situation.

Further, as with AMETEK.XLS {Ametek}, the number of weeks of data provided in this example is comparatively small when the high noise factor in stock prices is considered. In actual use, much more data and more extensive use of the kind of pre-processing we did with ROC, Momentum, and Moving Average Oscillator could help in establishing more sophisticated neural network models for stock or commodity trading.

Most important though is the complexity of the system being modeled by the neural network. If a system has a well-defined structure with clear relationships between data, then a neural network will do well when modeling it. If a system is completely unordered and random, or its relationships are much weaker than the noise that is present, then a neural network will not be able to model it successfully.

Important Points:

> Multi-row data windows are an important technique to present time structured (or any other inter-related) data to a neural network.
>
> Developing and providing intermediate processing results can help the neural network build more sophisticated internal models.
>
> Predictive accuracy is related to the complexity of the system being modeled and the amount of “noise” present in the system. In the worst case, random systems cannot be modeled by neural networks.

5.9 Shape Recognizer — SQUARE.XLS {Square}

SQUARE.XLS {Square} will demonstrate the use of Neuralyst’s two-dimensional capabilities for analysis and pattern matching through the recognition of squares as distinguished from other shapes. This is a simplified, but highly relevant example, derived from the problem of recognizing shapes in applications such as robotic vision, satellite image processing, or optical character recognition.

In these and other similar applications, image data is organized as a rectangular grid of values, where each grid position is called a *pixel*, or picture element. A photograph or other image may be *scanned* (another term is *digitized*) and converted to pixels by imposing this grid over the image and converting the amount of light that is present at each grid position into binary, discrete or continuous values. The greater the number of pixels, the greater the scanning resolution, and the more accurate a representation the scanned image will be of the original image.

For medium resolution image processing, it would be common to see grids of 100 by 100 to 250 by 250 and with pixels having 16 to 256 discrete values. For high resolution image processing, grids of 500 by 500 to 2000 by 2000 and pixels having 1000 to 4 billion (essentially continuous) discrete values are possible. The resolutions of your computer monitor or TV screen are representative of the high-end of medium resolution or the low-end of high resolution displays.

In SQUARE.XLS {Square} pattern data has been entered into the worksheet as symbolic values in an input grid that is 6 by 6; obviously low resolution, but the principles remain the same. This data consists of alternate examples of squares in various positions in the field, and assorted odd-shaped, but otherwise regular, “blobs”. The neural network will be trained to symbolic outputs, indicating whether the shape is a square or is a blob.

In order to train a neural net on image data we will need to have it accept inputs in a two-dimensional form. We will do this by expanding on the iterated data window technique used in DJIA.XLS {DJIA}, by increasing the value of Rows to Shift per Pattern in the Set Rows command from 1 so that the number of Rows to Shift equals the number of Rows per Pattern. This technique may be called *Stepped data windows*.

This example will also make use of *input noise*. Input Noise is one of the parameters that can be set by **Set Network Parameters**. Setting moderate values of Input Noise provides for slight variations in the value of training data between each training epoch. Moderate values of input noise encourage the neural network in the development of generalizations. Large values of Input Noise are usually not useful.

To run this example:

1. **Init Working Area** — starting at **R1**
2. **Set Rows** — **6** through **90**, 6 Rows/Pattern, 6 Rows/Shift
3. **Add Input Columns** — **C** through **H**

4. **Add Target Columns — J** and **K**
5. **Add Output Columns — M** and **N**
6. **Set Mode Flag Column — P**
7. **Set Mode Rows — 90**, Set Symbol Row
8. **Set Network Size** — 4 Layers,
   36 neurons on Layer 2,
   9 neurons on Layer 3
9. **Set Network Parameters —** Input Noise to 0.3,
   Epochs per Update to 5

Train the neural network with this configuration. After a short time, Neuralyst will stop training. Try testing the neural network on the additional cases given. You will find that Neuralyst has learned to distinguish squares from other shapes!

Try seeing what happens without Input Noise. Train with Input Noise reset to 0 a few times. While Neuralyst will still distinguish most shapes, it will not train as consistently nor perform as reliably. Try increasing Input Noise to larger values, up to 1. How well does the neural network train or perform under these conditions?

In addition to encouraging the neural network to develop generalizations, moderate values of Input Noise may also be used help keep a neural network from getting trapped in local minima (see Section 5.1).

You have probably noticed that this example uses a four layer network. Try multiple runs with only three layers (36 hidden layer neurons). You will find that Neuralyst gets the right answers most of the time, but not as often as with the four layer network suggested. Extra layers can improve a neural network's generalization capability, particularly when there are complex features in the data.

The technique we used in this example, iterated or stepped data windows, is primarily used for conserving data space and retaining data in formats familiar to the user. The neural network itself is not given any information about these organizational details when it is being trained or tested. Neuralyst presents all data items, regardless of their position in time or space, consistently to the neurons of a neural network so that any relationships between different neurons are developed as part of the training process.

As a matter of fact, this example could have been presented so that each grid of 6 by 6 was presented as a single row of 36 values with the same results. Similarly, three-dimensional and higher-order data structures can also be presented to Neuralyst by remapping them into a one- or two-dimensional format.

Important Points:

> Use stepped data windows to build neural networks to analyze two-dimensional problems.
>
> Use moderate values of input noise to improve training consistency and encourage generalization by the neural network.
>
> Use moderate values of input noise also to help keep neural networks from getting trapped in local minima during training.
>
> Adding more neural network layers can improve analysis capabilities when complicated relationships are present in the data.
>
> Multi-dimensional data structures can be presented in many ways to a neural network; the neural network will develop the necessary structural relationships.

Chapter 6
Advanced Neuralyst Topics

6.1 Input and Output Value Ranges

If you have read Chapter 3 carefully, you will be aware that neural network outputs can only range from 0 to 1. If you examine the sigmoid that is the default activation function for every neuron in the neural network, you can see the reason for this is a mathematical consequence of the way the sigmoid function works. In fact, a neural network also works best when its inputs range from 0 to 1, though the limitation here is not as absolute as for the outputs. While the examples we've shown used many inputs and outputs that complied with this range restriction, there were also instances where that wasn't true and Neuralyst still worked. In fact, if none of the values held to the restricted range, Neuralyst would still have worked.

So how does Neuralyst deal with these unrestricted ranges when neural networks work best within the restricted range of 0 to 1? Basically, Neuralyst pre-processes all input and output data and performs offset and scaling operations to match the actual ranges to the range 0 to 1. For example, if an Input column has values that range from 3 to 11, first 3 is subtracted so that the new range is 0 to 8, then each value in the new input range is multiplied by 0.125 (the reciprocal of 8) so that all values now fit in the range 0 to 1. In the case of outputs, the reverse process takes place.

In all cases, you are able to use numeric values that make sense or are convenient to your problem. Neuralyst will adjust the data in the input or output direction so the neural network is able to operate within its best value ranges.

Neuralyst also offers a variation of this scaling process through the Scaling Margin parameter in **Set Enhanced Parameters**. Scaling Margin causes the input and output data to be mapped into a smaller range than 0 to 1. A setting of 0.1, or 10%, in the Scaling Margin parameter causes an extra 10% to be reserved, 5% each at the bottom and top of the range, so the input and output data are now mapped into the range 0.05 to 0.95.

The allocation of Scaling Margin serves at least two functions. First, there are some problems, particularly those where the inputs and outputs are linear and continuously variable, that are better modeled when the activation function is restricted to output in the linear portion of its transfer function than the saturated, or non-linear, portion of its transfer function. Second, there are instances when new input or output values may be expected to be above or below those used to train or test the neural network model. Reserving some headroom allows some continued utility to the neural network model under these conditions, as long as the variant input or output values are not excessive.

Neuralyst also offers another extension to data representation by allowing the definition of symbolic input and output values. By defining a Symbol List for each Input or Target column, the Symbols are taken in order and interpreted for you. For example, if the Symbol List is defined as <RED, GREEN, BLUE>, then the range is divided into three equal subranges of 0 to 0.33, 0.34 to 0.66, and 0.67 to 1. An input value that is set to BLUE will be entered as 0.83, the center value of the 0.67 to 1 subrange. An Output value that is 0.95 will be found to be in the BLUE subrange and reported as BLUE. All the details of mapping Symbols to subranges and interpreting values to Symbols are all managed by Neuralyst for you.

A final issue regarding input and output value ranges are the defined minimum and maximum values for Input and Target columns. Each time that a neural network model is trained, the minimum and maximum values for each column define the range for that column that must be mapped into the neuron input or output range of 0 to 1, or subrange reduced by the Scaling Margin. If new data is later added to the neural network model that has a lower minimum or higher maximum in that column then that will cause a shift in all the values previously used to train the network. For any significant shift in the minimum or maximum values, this will invalidate the training of the neural network.

The MIN scale row and MAX scale row mechanism allows the actual minimum or maximum values at the time of training to be recorded and used to fix the scaling range to the correct range that was used for training. The Scaling Margin parameter will extend the set range by the designated amount allowing values that fall within the headroom region to be accepted, even if new values are added which exceed the stated range. If the MIN scale row and MAX scale row are set and values fall outside the Scaling Margin modified range, then the neural network will reject those values.

6.2 Setting Network Size

Setting the optimum network size for a problem is an intriguing and sometimes difficult process. The number of inputs and outputs are automatically defined by the structure of the problem. The number of hidden layers and number of neurons in each hidden layer are left to you (within the limits of Neuralyst's capabilities).

It has been shown that an arbitrarily large three layer (that is one hidden layer) backpropagation network can approximate just about any real-world mathematical function to an arbitrary degree of accuracy. If this is true, then why does Neuralyst allow as many as six layers (that is four hidden layers)? This is because in the three layer case the number of neurons in the single hidden layer may have to be so large that the neural network is impractical or impossible to implement. Multiple hidden layers allow greater flexibility for generalization (internal organization and model development) during the learning process and can significantly reduce the need for large numbers of neurons.

So how do we determine the number of hidden layers? There are no real rules regarding this, only rules of thumb. Generally, the more complex the problem and the more inter-related you perceive the problem characteristics to be, the more layers will be needed. It is best to start with three or four layers. Add additional layers only as the training proves to be difficult or impossible or as the predictive ability of the neural network proves unsatisfactory.

How do we determine the number of neurons in each hidden layer? Again, there are no real rules regarding this, just rules of thumb. Generally, it is best to have a pyramidal shape, that is have the greatest number of neurons in the initial layers and have fewer neurons in the later layers. A good working range for the number of neurons in each layer is to have a number from mid-way between the previous and succeeding layers to twice the number of the preceding layer. For example, if there were 12 neurons in the previous layer and 3 neurons in the succeeding layer, then a working range for the neurons in the intermediate layer would be about 6 to 24.

Often setting network size is a process of adjustment and iteration. You need enough layers and neurons for characteristics to be identified and generalizations to be made, but too many layers and neurons will be costly in computation time.

A methodical process that is often successful is to increase the size of the network (layers and neurons) in large jumps and look for successful training, then decrease the network size incrementally and observe the predictive accuracy.

The Neuralyst Genetic Supervisor can be used to automate the neural network optimization process using genetic technology. See Chapter 7 for a further discussion.

6.3 Learning Rate, Momentum, and Training Tolerance

The settings of Learning Rate and Momentum control the way in which the error is used to correct the weights in the neural network for each training case. Some settings may cause surprising behavior.

When Learning Rate is set to high values (close to 1), there is some possibility that you will see unstable behavior. This is often evidenced as wildly varying values of RMS Error (remember that neural networks also undergo learning plateaus which may show small increases of RMS Error for brief periods; we are not talking about these). As the Learning Rate is set lower the possibility of unstable behavior is reduced. Generally, when Learning Rate is set to 0.25 or less, you will not see any unstable behavior. However, lower values of Learning Rate will result in longer training times.

A more aggressive, though workable, approach is to set the Learning Rate high and to decrease it if you see any unstable behavior. This is essentially what happens when you use the Adaptive Learning Rate mode in the **Set Enhanced Parameters** command. A more conservative approach is to start with Learning Rate low and to try increasing it if training is taking too long.

The higher the value of Momentum, the greater the percentage of previous errors is applied to weight adjustment in each training case. For example, when Momentum is set at 0.5, then 50% of the weight adjustment will be due to the current error and 50% will be the weight adjustment applied in the previous case.

This means that any weight adjustment due to the error from one training case will have a continuing effect with an exponential decay. For example, consider a training case *t*; with Momentum set to 50%, the weight adjustment will be portioned as just described. In the next training case, 50% of the weight adjustment will be due to the error from *t+1*, 25% will be due to the error from *t* and the remaining 25% will be due to the previous cases. In the third training case, 50% of the weight adjustment will be due to *t+2*, 25% will be due to *t+1* 12.5% will be due to *t* and 12.5% from previous cases. As each case is considered, then the effect on weight adjustment of each previous case is halved (or reduced in proportion as determined by the Momentum setting if it is not 0.5).

In the special case when Momentum is set exactly at 1, then 100% of the previous error is used for weight adjustment. In the very first training instance, there are no previous weight adjustments, thus the first weight adjustment will be 0. Since the next weight adjustment takes 100% of the current weight adjustment, that will also be 0. This continues, regardless of how many training instances are considered. This is why Momentum must be less than 1.

In the special case when Momentum is set exactly at 0, then 0% of the previous error is used. Therefore, each weight adjustment is applied as 100% of the weight adjustment due to the current training instance;

no previous values are included in the adjustment. This is a workable setting.

Setting Momentum to higher values helps to smooth out the training process and prevent unusual cases from throwing the training off track, while continuously correcting for consistent errors. Setting Momentum lower may be appropriate for data which is more regular and smoother or data for which the relationships to be learned are relatively simple. It is generally necessary to experiment with different values of Momentum to find an appropriate value for a particular problem.

The Neuralyst Genetic Supervisor can be used to automate the Learning Rate and Momentum optimization process using genetic technology. See Chapter 7 for a further discussion.

The setting of Training Tolerance is used primarily to determine how much training the neural network undergoes. When Training Tolerance is set to a number, for example 0.4, Neuralyst will continue training the neural network until the output of the neural network for every training case is within 40% of the target range for every target value. Thus, if the target range has values from 100 to 300, then 40% of the target range is 80, or 0.4 x 200 (300-100). Thus every output value must be within 80 of the target value before Neuralyst will stop training.

It is not usually useful to have the Training Tolerance set to greater than 0.5. In the case of binary (0 or 1) values, Training Tolerance set to 0.5 means that the outputs must be less than 0.5 to be considered a 0 and greater than 0.5 to be considered a 1. While this is sufficient for binary values, and proportionately tighter Training Tolerances would be appropriate for multi-valued output cases, there are times when you may want to set Training Tolerance tighter than strictly needed to distinguish between the number of distinct outputs or to be within some error allowance for continuous outputs.

Generally, neural network output accuracy on testing or running data will be less than accuracy on training data. So, if a neural network were trained to a Training Tolerance of 0.5, outputs, for example, that should be 1's would range from 0.5 to 1. If it were then run on testing data, the outputs that should be 1's might now range from 0.3 to 1. This would result in some outputs being treated as 0 when they should have been 1. Training to a tighter Training Tolerance, for example 0.4 or 0.3, helps alleviate these marginal conditions, in effect providing a "guardband" between values that should be distinct or an extra margin for error for values that are continuous.

However, setting Training Tolerance too tightly (close to 0) may create additional problems. The first problem is a matter of practicality. Tight values result in longer training times. The second problem is more subtle. Since training time is increased, the opportunity for overtraining is also increased.

Overtraining occurs as the neural network is forced to match the target values more exactly. During this process it devotes capacity to learning the exact values of the training data rather than to generalization from the training data. If insufficient capacity is available, then the Training Tolerance is not achievable and the neural network will never stop training. If sufficient capacity is available, then the Training Tolerance will be achieved, but testing or running accuracy will suffer.

It is interesting to observe the behavior of the RMS Error generated by the neural network on training data versus testing data as the neural network trains. This can be done by setting the Error Limit value in **Set Network Parameters** to a moderate value, for example 0.1, training the neural network, and then performing a **Plot Training Error**. Typically you will see a reduction in the RMS Error plot for the training data, with the rate of reduction changing over time. If the neural network begins to overtrain, you may eventually see it as a long plateau in the RMS Error plot for training data; in some cases, the plot may begin rising. However, you will generally see it much earlier as an increase or rise in the RMS Error plot for testing data.

The Error Limit setting and the **Plot Training Error** command are powerful tools for detecting the onset of overtraining; though at the expense of increased processing time.

6.4 Learning, Weights, and Multiple Solutions

Learning takes place in neural networks through a weight adjustment process that is driven by the error developed by the neural network in each training instance. In a sense, you can visualize each weight set as defining a point on a globe (using weights as latitude and longitude on the globe). A neural network weight set that produces the desired outputs then corresponds to a special location on that globe. The training process is then similar to the game of "hot-or-cold"; as each training instance is presented, the neural network may be told that it is "cold" (or "hot"), in which case it moves a lot (or a little) across the globe by adjusting its weights a large (or a small) amount. As the neural network gets closer to the right location, it will get "hotter" until it moves just enough to find the spot.

The weight adjustment process for a given weight of each neuron is dependent on the prior value of the weight, the actual output, the desired output and the related input for that neuron (see Section 3.4). If the weights for all neurons were set to the same initial value, then all neurons having the same output value and the same input values in a training instance would have those weights adjusted by the same amount (since the weight, input, output and desired output would be the same). This is not the most desirable behavior and under some circumstances can lead to a failure to learn (like going around in circles). By setting the initial weights of a neural network to random values, this undesirable behavior is avoided. However, this randomization process raises another issue.

It is possible that more than one weight set will satisfy the input data and training constraints (like looking for cities with population over 1 million that have English as the primary language — New York, Los Angeles, London, and others would fit). In such cases, the initial value of the random weight set (starting point) could change the solution (city) the neural network finds each time. It is possible that copying a randomized weight set, saving it, and copying it back every time you wished to restart training could minimize this problem. However, reducing, extending or even changing the sequencing of the training data could change the solution found as well.

The existence of these multiple solutions may be disturbing to you since they can manifest themselves as different predictions or behavior by the neural network. However, within the constraints of the problem presented, any of these are perfectly satisfactory solutions, so the neural network has no reason to prefer one over the other. If you have such a preference, then you need to provide more training data.

Setting Training Tolerance to high values also has the effect of creating multiple solutions artificially. Setting any Training Tolerance is analogous to drawing a circle around the special location (or locations) that is desired. Any given point in that circle would be a satisfactory solution. If the Training Tolerance is low enough (circle is small), most weight sets (corresponding to points within the circle) would behave in much the same way. If the Training Tolerance is high enough (circle is large), some weight sets may behave differently than others.

6.5 Some Causes of Poor Results

If a neural network is difficult to train, tests poorly, or if its real-world performance doesn't match its training and testing performance, what can be done? The answer to this question lies primarily in having sufficient experience with neural networks to understand their capabilities and limits. The more neural networks you design and use, the better you will become at getting the best results from them. In many ways this is as much an art as a science. Until that experience is gained, here are some considerations.

First, the neural network needs to be properly sized for the problem. As discussed before (see Section 6.2), the number of layers and number of neurons are under your control: too small a network and the neural network may not be able to learn the problem; too large a network and the neural network takes longer than necessary to train. You need to observe the training behavior and test results to make these judgments.

Second, also as discussed before (see Section 6.3), the neural network may be overtrained. One way, already mentioned, to avoid overtraining is to set a looser Training Tolerance (so that training stops sooner). A second way is to set one of the cutoff limits so that

Neuralyst will stop training after a certain number of epochs, after a certain amount of time, or if RMS Error starts increasing.

Another technique to prevent overtraining, particularly with a limited training set, is to modify the training data from presentation to presentation so that some *noise* is present. That is, vary the training data slightly with successive presentations so that the network doesn't recognize the exact values presented but rather the general pattern. This can be done by setting Input Noise to small values. You must choose the amount of noise added so that it is sufficient to encourage generalization without overwhelming the underlying relationships.

Another way you can address this is with more training data, enough so that the neural network is forced into generalizations rather than specifics. When preparing more data, be aware that some underlying factors may not be constant and may have changed over the extended training set (or even in the original training set); this is particularly relevant for time-based data. If increasing the data set also increases the amount of variation of such factors, then there needs to be enough cases representative of each combination of factors for the neural network to be able to distinguish between these. For example, it may be very relevant to a fundamental analysis based model of a given stock that the most current rise in sales was due to war-driven revenues; however, without one or more additional examples of this effect from other wars, the neural network may not be able to learn the reason for the rise.

When the decision is made to increase the amount of data used for training after experiencing poor performance, another decision must also be made. Should the training be done incrementally, that is, resume training with the existing weight set while adding the new data, or should the training be restarted from a completely “blank” neural network? The answer to this really depends on the problem and its relationships.

If there is only one weight set that the neural network can develop to satisfy the relationships inherent in the input data and targets, then the incremental training and the blank-slate retraining will both achieve the same general solution. If there are multiple weight sets that the neural network can develop, due to the existence of multiple sets of relationships that are consistent with the data or due to a loose setting of Training Tolerance which might allow several solutions that are correct within the tolerance range, then the weight set actually developed by the neural network will depend on how the training was done.

While the weight set developed by the neural network in any case will satisfy the data presented and constraints established during training, it is possible that some other weight set, if it exists, might be a better predictor during testing or running (see Section 6.4). Different neural network weight sets (and thereby different predictive behavior) may develop when trained incrementally as compared to training from a blank-slate. This is often an indicator that the Training Tolerance has been set high enough that multiple solutions are acceptable. This may also be an indication that one or more factors or relationships changed over the initial training period versus the incremental periods. This may not be a problem unless you don't have enough data representative of these changing factors or relationships (see the next paragraph). Usually, the only way to verify the existence of this phenomenon in your problem is to try training with different increments and observing the resulting predictive behavior.

In some cases, results may be poor because you haven't chosen a complete set of data representative of the conditions that are relevant. This was alluded to earlier in terms of the number of cases in the training data being sufficient to be representative. But this also applies with regard to the different input types that you choose to present. For example, if you were to design a neural network for predicting customer activity in a department store, you would probably find that it would be more accurate if some inputs indicated Federal and local school holidays, though your first attempt to build the model might only have considered the day of the week.

In fact, the success of a neural network depends to a large extent on your ability to find the relevant types of input data and the manner of their presentation. It is possible to present all the data types which you perceive to have any relevance. In this case, the neural network will, if it has enough capacity, learn which data types are really relevant and which it can ignore in making a prediction. However, this raises the cost of obtaining and maintaining the data for training the neural network. Further, it requires greater computer resources in memory space and computing time in order to process the increased neural network size. Realistically, you must select just a few data types that you think are most relevant and include or exclude a few types over time as you measure the performance of the neural network.

In many cases, input data can be improved through restatement or intermediate processing. A neural network can develop many internal operations and correlations on its own; however, it makes no sense to train the neural network to perform, for example, a moving average, when Excel can compute it more exactly and more easily. Performing these intermediate computations can reduce the training time, free neurons for generalizations and, in many cases, facilitate the process of generalization. (Software programmers are familiar with this last phenomenon; the choice of data structures and programming languages makes some operations much more or much less difficult to implement even though they can all be expressed ultimately.)

The neural network targets deserve just as much thought as the inputs. In some cases, for example, rule reproduction, the targets are fairly obvious and easy to state. In other cases, particularly with regard to predictive models, the targets will require some thought in order to achieve the best statement. (Oracles throughout human history have been notorious for requiring very careful statement of the questions put to them!)

For example, the target for a technical analysis based stock model could be stated in many ways: a future price level, a future price change, a future price movement (up versus down), the number of days to an up or down move, or the volume of shares to be traded prior to an up or down move. Which of these is the best target depends not only on what you want to achieve but also on the kind of data available to be presented to the neural network. A neural network cannot make any oracular predictions (despite the comment in the last paragraph), it can only develop predictions from relationships that are present in the data put to it.

These are just a few points of consideration if you find yourself having a difficult time solving your problem with a neural network. While there are many other points that could be made, this will help you over a large number of tough spots.

6.6 Experimenting with Enhanced Neural Networks

So far our discussion has focused on the standard backpropagation neural network, as described in Chapter 3. Neuralyst also provides several ways to modify the standard backpropagation neural network. These can be grouped into the categories of selectable neuron activation function, the ability to force weights to zero if they become “insignificant”, and the ability to adapt learning rate dynamically in proportion to the error of the neural network output. Let’s see how these enhanced functions can help you to develop a better neural network.

Neuron activation functions cause a “decision” to be generated from a neuron. Thus changing the neuron activation function or its activation parameters will change the nature and characteristic behavior of the decision process. Six basic activation functions are available in Neuralyst: Augmented Ratio, Gaussian, Hyperbolic, Linear, Sigmoid, and Step. The behavior of all of these functions, except the Step function, can be further adjusted through a Function Gain parameter.

The Hyperbolic, Linear and Sigmoid functions are similar to each other. They all generate inhibitory outputs for negative inputs and excitatory outputs for positive inputs.

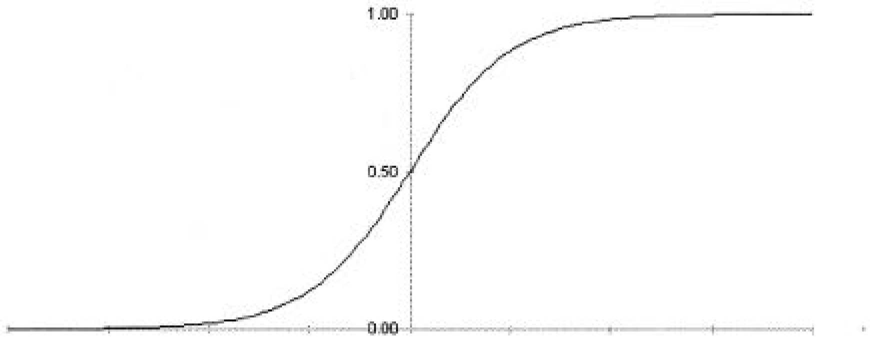


Figure 6-6 Sigmoid Function

The Sigmoid function typically has a narrow region about zero wherein the output will be roughly proportional to the input, but outside that region the Sigmoid function will limit to full inhibition or full excitation. Thus a Sigmoid function is a switch with an intermediate range where it can be discriminating.

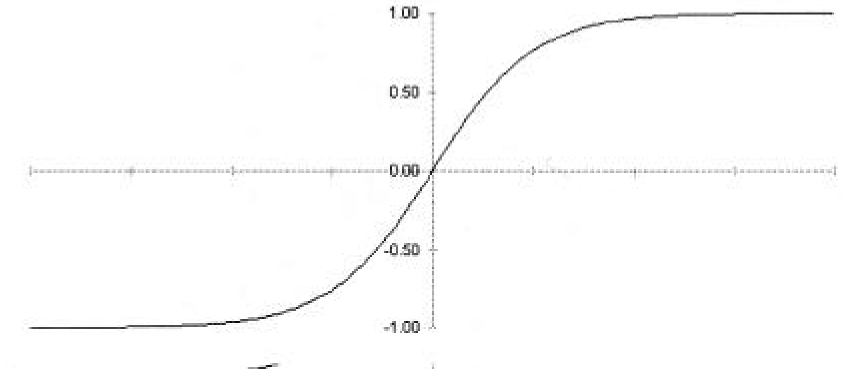


Figure 6-8 Hyperbolic Function

The Hyperbolic function is shaped exactly like the Sigmoid function, but it ranges from -1 to +1 rather than 0 to 1. Thus it has the interesting property that there is inhibition near 0, but values at either extreme will be excited to full levels, but in opposite sense. A Hyperbolic function is also a switch with an intermediate range where it can be discriminating.

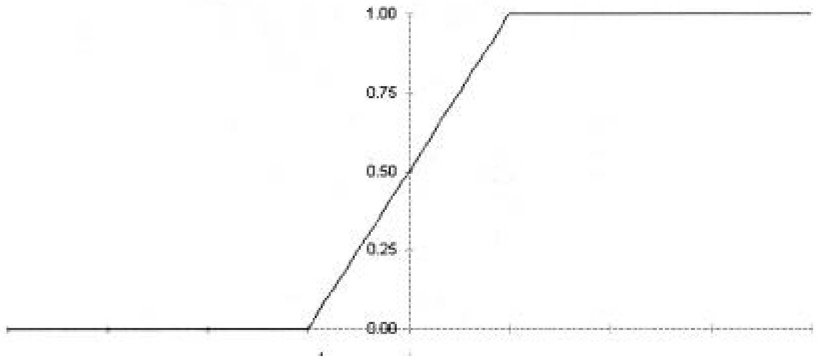


Figure 6-7 Linear Function

The Linear function always generates outputs which are proportional to the inputs, up to the level of full output. A subtle but important distinction is that the Linear function is stepped, at the point when it transitions from proportional output to full output, whereas the Hyperbolic and Sigmoid function are always smooth, that is differentiable. Differentiability of the neuron activation function is an important factor in getting consistent backpropagation training behavior.

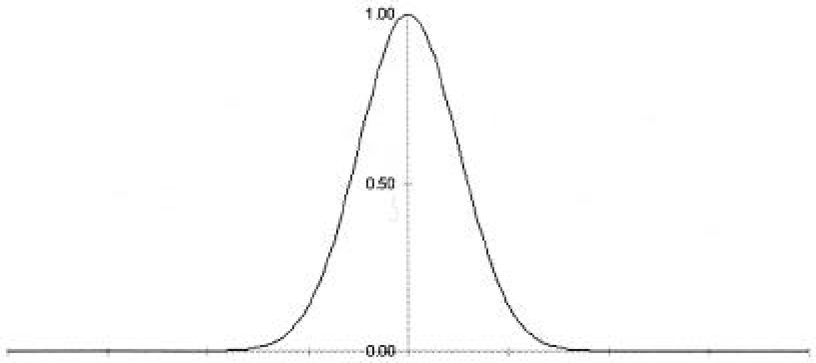


Figure 6-9 Gaussian Function

The Gaussian function is an interesting variation on the other functions. It is derived from the equation which generates a normal probability distribution and it has a central peak and low tails at both ends. This results in excitation close to zero inputs and inhibition as the inputs vary more significantly from zero.

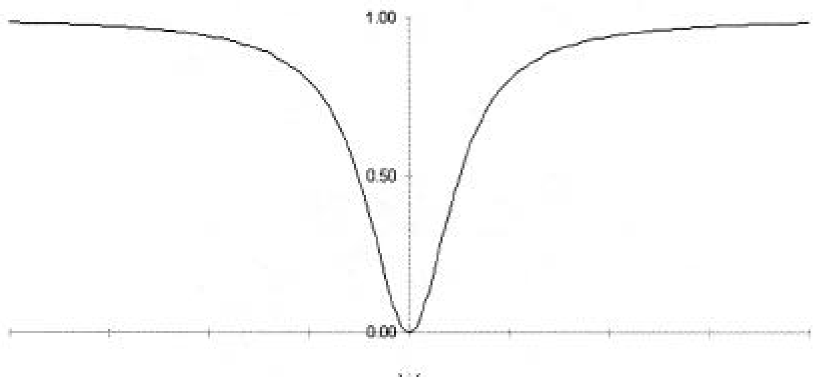


Figure 6-10 Augmented Ratio Function

The Augmented Ratio function is an upside-down version of the Gaussian function. It is defined by the equation

$$y = \frac{x^2}{1 + x^2}$$

which is also known as the augmented ratio of squares. It has a central valley reaching 0 and high tails at both ends. This results in inhibition near zero and excitation as the inputs vary significantly from zero. The Gaussian function and Augmented Ratio are both smooth (differentiable).

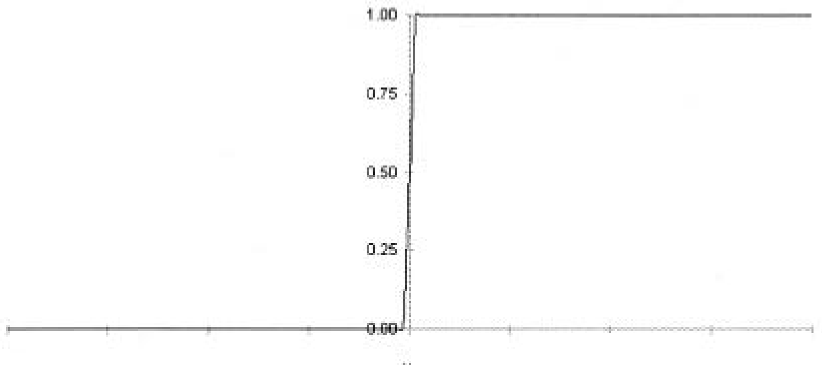


Figure 6-11 Step Function

The last function, Step, is not generally very useful, but it can be interesting to observe its behavior for simple problems. It was actually the original activation function used in early neural networks called Perceptrons. The Step function basically converts any negative input into a fully inhibitive output and any positive input into a fully excitatory output. This has the behavior of a switch; there is no fine discrimination. It did not take long to reach the limit of capabilities of Perceptrons and neural networks progressed beyond them.

Generally the Sigmoid function is far and away the most useful and the Step function is the least useful. But an examination of the characteristics of your own data and the desired decision function behavior may lead you to try a non-Sigmoid function. Experiment!

Frequently inputs or connections will have a low contributory effect on the output of a neural network. When this happens, the backpropagation algorithm will change the weights over time to be close to zero. However, there will still be some instances of excitation which will cause the weight to jitter around zero. When this happens, it increases the noise or uncertainty associated with an output. Neuralyst has the option to select a threshold and to force weights which fall below this threshold to be kept at a zero value. Forcing a weight to zero is equivalent to breaking a connection in a neural network.

It is usually best to set this mode after some initial training. Otherwise it would be possible for weights which start close to zero, but which would have been adjusted to a respectable value, to be forced to zero unnecessarily. Another approach is to complete training, then do one epoch of training with this mode set. This can help to clear the neural network of meaningless connections. When this mode is set, it is also frequently interesting to perform an **Unpack Weights** command and use the resulting display to correlate the zero weights with your input data.

It is important to be judicious in your use of this mode. It is possible that some critical analysis may hinge on the resulting fine shadings or distinctions generated by these low value weights! Always test your results and evaluate the quality of predictions before and after setting this mode.

In some instances, it may be difficult to find a setting of Learning Rate that will allow learning to occur but which doesn't take an inordinate amount of training time. This can happen because large settings of

Learning Rate cause too much adjustment to the weights during each training epoch, causing the neural network to jitter around the right area, while small settings of Learning Rate make too little progress during each training epoch.

Neuralyst provides an option to enable an Adaptive Learning Rate to address these situations. With this mode set, the Learning Rate parameter is ineffective. Instead, Neuralyst will set the Learning Rate to be proportional to the RMS Error generated during a training epoch. Thus when a neural network is far away from being correctly trained, the RMS Error will be high, and the Adaptive Learning Rate will be at a maximum. As the RMS Error is reduced, the Adaptive Learning Rate will be reduced proportionately. When the RMS Error is very small and the neural network is on the verge of completing training, the Adaptive Learning Rate will be at a minimum level still able to achieve effective learning.

Note that it is not the best solution to enable the Adaptive Learning Rate mode in all cases, since there are still many instances when a large setting for Learning Rate will train a neural network perfectly well. Setting the Adaptive Learning Rate in these instances will still achieve correct and possibly even better training, but it will take longer to complete the training, since the Learning Rate will be smaller than could be useful.

6.7 Excel Charts and Neuralyst

The use of Excel's built in charting functions can greatly enhance the utility of Neuralyst. Neuralyst provides the **Histogram Weights** and **Plot Training Error** commands (see Sections 8.2.8 and 8.2.10), but Excel charts can be used with Neuralyst in many other ways.

One use is for the graphic presentation of your data. Thus, stock price data can be shown as a High-Low-Close price chart, marketing data can be shown as pie or bar charts, technical data can be shown as line or scatter plots, and so on. Input data, target data and neural network outputs can be shown as separate lines or regions, line extensions or on their own charts.

A second use is to help you visualize the match between targets and outputs after training. This is particularly useful with regard to different settings of Training Tolerance. Comparison plots of targets versus outputs can show you how well the neural network may be learning and also let you see whether or not tighter Training Tolerances may be needed.

Charts can also aid in visualizing the behavior of your neural networks. One technique, discussed in the example `FIZZY.XLS` {`Fizzy Cola`} (see Section 5.6), is to generate a response curve for individual inputs to the neural network. This is done by holding all inputs except one constant and then varying that one evenly across its input range. The resulting response curves generated in this way can give you a picture of how each input affects the outputs both in direction and sensitivity.

Another technique is to plot the accuracy achieved on test data for an increasing number of training epochs (by decreasing Training Tolerance — see Section 6.3). If this technique is combined with trials of different numbers of network layers and changing numbers of hidden layer neurons, then it can be helpful in visually identifying the optimum neural network configuration for your problem.

Take advantage of Excel's plotting capabilities. They can help improve your presentation of the data as well as improve your understanding of a given neural network and its behavior.

Chapter 7
Genetic Optimization of Neural Networks

7.1 Genetic Technology

A new feature of Neuralyst is the inclusion of a Genetic Supervisor for enhancing the development of a given neural network model. Normally, a user of Neuralyst must experiment with the characteristics of a desired neural network model, adjusting the number and types of Input columns, the neural network configuration, and the neural network parameters in order to determine the characteristics which best produce successful predictions with the minimum amount of error or training time or both.

The Genetic Supervisor will optimize the Input column set and training parameters for a Neuralyst neural network model so that subsequent usage or adaptation of the neural network model will perform well as a successful predictor with a minimum amount of training. This is done in an automated fashion, but at the expense of a lengthy run for moderate size problems.

The Genetic Supervisor uses a special type of optimization technology known as *genetic algorithms*. Just as neural networks are models of biological neural networks that have the properties of adaptation and inference; genetic algorithms are models of a selective evolutionary process which develops superior entities from a population of entities. The genetic algorithm used in Neuralyst attempts to select the best subset of data from that provided as input, configure the best neural network that will train with the data, and adjust the various parameters that control the neural network to optimum points.

While there are many methods for determining the optimum solution to well-defined problems; there are very few methods for finding optimal solutions to unstructured, poorly understood, or partially unknown problems. Neural networks and genetic optimization are two of these limited set of methods.

7.2 Operation of the Genetic Supervisor

Although genetic algorithms are directly modeled after biological systems and behavior, the specific terms used in these two disciplines are different. The Neuralyst Genetic Supervisor uses terms from the literature of genetic algorithms to describe its parameters and behavior. Due to the close relationship to biology, the language and terms used to describe genetic algorithms can be defined in terms of their correspondance to genetics terms.

In biology, specific genetic terms are used to express the desired meaning. The *chromosomes* of a biological genetic system correspond to *strings* in an artificial genetic system. Chromosomes are composed of *genes* which take on values called *alleles*. Examples of genes are those for eye color or height; examples of alleles are blue and tall. The corresponding terms in artificial genetic systems are *features* and *values*. For example, in the genetic system in Neuralyst two features are input column name and learning rate; these features may take the values A and 0.85, respectively. The entire genetic package of chromosomes is called a *genotype* corresponding to the entire package of strings called a *structure*. In the Neuralyst genetic system, there are three strings, one for Input columns, one for network configuration, and one for network parameters; the combination of all three representing a structure which can define a neural network configuration. The expression of a genotype as a biological entity is called a *phenotype*, which corresponds to the expression of a structure as a *candidate solution*.

All the values of a structure represent the neural network characteristics that uniquely define a candidate solution in the space of possible solutions. The Neuralyst Genetic Supervisor evolves successor populations, or *generations*, from a limited population of initial candidate solutions. It does this by treating the inclusion or exclusion of each column of data in the full Input column set as features; the number of layers and the number of neurons per layers as features; and the control parameters of each neural network as features.

These features are then varied in each new generation with the resulting structure evaluated in terms of neural network *fitness*. Each structure in the generation is evaluated and judged by either the lowest RMS Error achieved after a fixed number of epochs or by the number of epochs taken to achieve a minimum point in RMS Error. These two measures represent neural networks that train to minimal error or neural networks which train with minimal epochs. These criteria can be applied to the set of training cases or the set of test cases.

If the structure representing a neural network successfully meets the fitness criteria selected, then the values of its features will be retained and bred with other structures. For each generation, the Genetic Supervisor generates a population of structures in one of two ways. All the structures of an initial generation and a certain number of structures in subsequent generations are created with features set to random values constrained within specified limits. Subsequent generations are created by cross-breeding the strings of successful structures or occasional mutations of randomly selected features of successful structures. Some or many of the weakest structures may be culled, these are replaced with new structures.

Through this evolution-like process, an optimal neural network can be developed. Note, however, that this process requires the training of many versions of the neural network to determine an optimal one; for neural network models that have large network configurations or have large data sets this can be a lengthy process — but so can biological evolution!

7.3 Structure Strings and Features

Each candidate solution is represented by a structure composed of three strings. The strings represent the selection of Input columns, the neural network configuration, and the neural network parameters.

7.3.1 Input Column Selection

Each Input column that is selected as part of a neural network model is included because the user believes that the data is correlated, positively or negatively, to the desired output. However, the data may or may not be in fact useful in predicting the output data. Further, even if useful, the data may be redundant with respect to other data already present. Thus the Genetic Supervisor treats each column of data as an individual candidate for exclusion or inclusion.

Only the set of Input columns is considered for optimization as it is presumed that a specified Target column and its corresponding output column are always a necessary part of the model. In the few cases where targets and outputs are redundant, then you will need to restrict the targets and outputs for best performance.

The Input column set is represented internally by a string of features which specifies the exclusion or inclusion of the Input columns for the structure. There is a feature for every possible column that can be included or excluded. There are two possible values for each feature, inclusion or exclusion. The user may select a nominal percentage from 1 to 100% which represents the average rate of inclusion. If the user wants to force all data to be included, then the user can set 100%. The default setting is 75%.

A percentage less than 100% does not mean that 100% of the Input columns will never be included; it means that achieving 100% inclusion will require cross-breeding and mutation to reach that level. Similarly, specifying 100% does not mean that there will never be Input column sets that are less than 100%; it means that achieving less than 100% inclusion will occur through mutation and subsequent cross-breeding.

7.3.2 Neural Network Configuration

The first and last layer of each neural network is automatically determined by the number of inputs and outputs defined by the candidate solution representing a neural network model. The count of included values in the Input column string represents the number of inputs. The number of outputs remains constant and is determined by the base neural network model. Within those constraints, the neural network can have from two to six layers and a wide range of variations in number of neurons per layer.

The neural network configuration is represented by a five feature string, where the first feature represents the total number of layers for the neural network and each subsequent feature represents the number of neurons in the corresponding hidden layer. An implicit rule is that if a lower layer feature has a value of zero neurons, then higher layer features must also be zero. The user can constrain the maximum number of layers and the maximum number of neurons per layer. The default maximum settings are 4 layers, that is an input layer, 2 hidden layers, and an output layer; with 30 neurons in the first hidden layer and 10 neurons in the second hidden layer.

7.3.3 Neural Network Parameters

The training behavior of a neural network is controlled predominantly by the user settable parameters for Learning Rate, Momentum, and Input Noise. The Learning Rate applies a greater or lesser amount of correction as a result of the error derived from a given case. The Momentum averages the current error to a greater or lesser extent with previous errors. The Input Noise factor causes the data to be perturbed from a slight extent to none to simulate real world noise or perturbations. These settings can affect the rate of training and the robustness of the neural network on test cases. The neural network training parameter features are represented by a three feature string, where each feature of the string represents the value of a particular parameter. The Learning Rate and Momentum can vary from almost zero to one, while the Input Noise can vary from 0 to 0.1. The user can constrain the minimum value of Learning Rate, the maximum value of Momentum, and the maximum value of Input Noise. The default values are a minimum of 0.5 for Learning Rate, a maximum of 1 for Momentum, and a maximum of 0.03 for Input Noise.

7.4 Population Management

There are two controls which determine the management of structure population. The first is total population Pool Size. The second is population replacement Pool Mode.

7.4.1 Population Pool Size

The population Pool Size controls the number of structures created or evolved in each generation. The number of structures remains constant, but the management of the structures is controlled by the population Pool Mode control.

Setting a large population pool provides a richer number of structures to test and evaluate to generate an optimal neural network. However, large pools will take greater amounts of time to test and therefore search. Setting a small pool has the reverse effect, each generation completes more quickly, but the pool is sparser and may take a longer amount of time to reach an optimal network.

Current research indicates that if processing resources are limited, corresponding to limited processing time, then the best strategy is to select a limited pool of 3-5 structures and to weed out the weakest structure at each generation. Conversely, current research also seems to indicate that if processing resources are unlimited, corresponding to unlimited processing time, then the best strategy is to pick as large a pool as possible. The default population Pool Size is set to 3.

7.4.2 Population Management Mode

The population Pool Mode control is a switch which sets one of three modes: Closed Pool, Immigration, or Emigration. These three modes represent different population replacement strategies from generation to generation.

With Closed Pool set, then the existing population pool will be evolved with no new structures ever introduced except through cross-breeding and mutation. With Immigration set, the population pool will be evolved and each generation entirely new structures will replace the weakest structures of the current pool with the remaining structures being cross-bred and mutated. With Emigration set, each generation the best members of a current pool will be emigrated to an entirely new population and cross-bred with the new population.

Closed Pool is similar to inbreeding in that weak structures are perpetuated through the transfer of their strings. For very small pools this is probably not useful; but for large pools, this may work fine. Immigration corresponds to the most versatile population management mode in that weak structures are continuously culled. It should work well for small or large pools. Emigration may be best for small pools, as it allows new structures to be tested rapidly and compared to the best current structures. In the case of large pools, emigration may also work well, but the characteristics of the best structures may be rapidly diluted. The default population management Pool Mode is Immigration.

7.5 Genetic Operators

Each time a population has been fully evaluated, each structure is ranked by its fitness. The structures are then evolved to generate a new population with each structure deriving its features from the previous generation such that the most fit structures of the previous generation have a higher chance of passing on their features in proportion to their ranking.

There are two genetic operations the user can control to determine the mechanism for passing features to the succeeding generation. These genetic operators are cross-breeding and mutation.

7.5.1 Cross-breeding

The primary genetic operator is cross-breeding which is determined by the Crossovers setting. Crossovers determines the frequency of intermingling of features on the same string to create new structures. Thus a setting of 1 means that two strings are crossed over at one point; a setting of 2 means that two strings are crossed over at two points, etc. The maximum setting for crossover frequency is 10.

As an example of cross-breeding, if Input columns **A**, **B**, **C**, and **D**, are selected for the base neural network model; then the string

X = <1,0,1,1> means that only Input columns **A**, **C**, and **D**, are included in this structure. If a second string, Y = <0,1,1,0>, from another structure is cross-bred with the first string, such that X is read first then crossed over at the mid-point to Y, then the resulting string for a new structure would be <1,0,1,0>, representing Input columns **A** and **C**.

Note that only the Input column string will actually be likely to see a high frequency of crossovers as the neural network configuration and neural network parameter strings are too short to intermingle more than once or twice. The default setting for Crossovers is 1.

7.5.2 Mutation

The secondary genetic operator for creating new structures is mutation. With mutation, structures and features are chosen at random, then randomly changed to new values. The Mutation Rate parameter sets the percentage of structures which will undergo a mutation rather than a crossover to create the new population.

As an example of mutation, if Input columns **A**, **B**, **C**, and **D**, are selected for the base neural network model; then the string <1,0,1,1> means that only Input columns **A**, **C**, and **D**, are included in this structure. If mutation is selected for this structure, and the third feature is mutated, then the new string would be <1,0,0,1>, representing Input columns **A** and **D**.

Mutations and cross-breeding are never mixed. A structure is either cross-bred or mutated. The maximum setting for Mutation Rate is 100%. The default setting for Mutation Rate is 10%.

7.6 Fitness Criteria

The evaluation of each structure is based on training or testing while comparing the best RMS Error level achieved or the least number of epochs. There are four modes which can be set: Train Epochs, Train Error, Test Epochs, and Test Error.

Train Error finds the structure with the least RMS Error when training each candidate solution up to the number of epochs specified in the Fitness Limit; it operates on training data only. Train Epochs finds the structure with the least epochs to achieve the RMS Error level set in Fitness Limit; it operates on training data only.

Test Error finds the structure with the least RMS Error in the test set data when training each candidate solution up to the number of epochs specified in the Fitness Limit; it operates on training data and testing data. Test Epochs finds the structure with the least epochs to achieve the RMS Error level of the test set data as set in Fitness Limit; it operates on training data and testing data.

The default Fitness Criteria mode is Train Error with Fitness Limit set to 100 epochs. This will optimize RMS Error while training each candidate solution to an epoch limit of 100.

7.7 Genetic Supervisor Tutorial

In Chapter 5, we trained a neural network to predict the possible investment potential of a NYSE stock, Ametek. We will now use that example to show you how to use the Genetic Supervisor.

Let’s recap the example: AMETEK.XLS {Ametek} contains fundamental stock data. The data is primarily organized on a per share basis. These are: sales revenue per share (Sls/Sh), cash flow per share (CF/Sh), earnings per share (Ern/Sh), dividends per share (Div/Sh), capital spending per share (Cap$/Sh), book value per share (BV/Sh), average price to earnings ratio for the year (Avg. P/E), relative price to earnings ration for the year compared to the overall market (Rel P/E), dividend yield (Div %), and the average price per share for the year (Avg $/Sh).

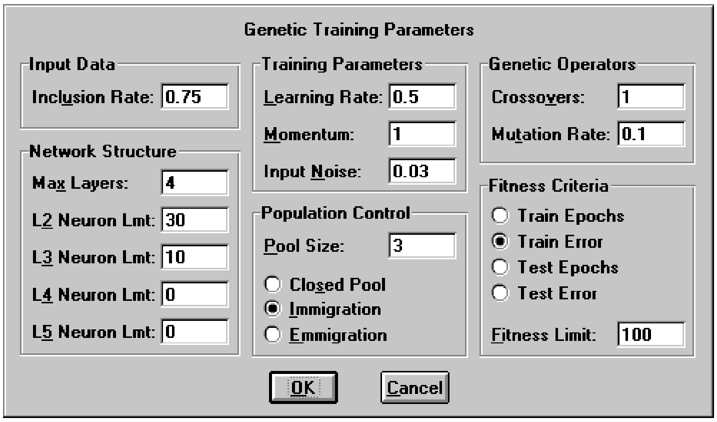
First set up the example in the same way as you did previously.

To run this example:

1. **Init Working Area** — starting at **V1**
2. **Set Rows** — **29** through **48**, 1 Row/Pattern, 1 Row/Shift
3. **Add Input Columns** — **C** through **L**

4. **Add Target Columns — N** and **O**
5. **Add Output Columns — Q** and **R**
6. **Set Mode Flag Column — T**
7. **Set Mode Rows — 48**, Set Symbol Row
8. **Set Network Size** — 3 Layers, 6 Hidden Neurons
9. **Set Network Parameters** — Epochs per Update to 10

At this point the neural network configuration is set exactly as it was the first time. Now we will allow Neuralyst to develop a more optimal neural network using genetic optimization technology. Select the **Set Genetic Parameters** command from the **Neural** menu. This will cause the Genetic Parameters dialog box to appear.



Genetic Training Parameters

Input Data
- Inclusion Rate: 0.75

Network Structure
- Max Layers: 4
- L2 Neuron Lmt: 30
- L3 Neuron Lmt: 10
- L4 Neuron Lmt: 0
- L5 Neuron Lmt: 0

Training Parameters
- Learning Rate: 0.5
- Momentum: 1
- Input Noise: 0.03

Population Control
- Pool Size: 3
- Closed Pool [✓ Immigration] Emmigration

Genetic Operators
- Crossovers: 1
- Mutation Rate: 0.1

Fitness Criteria
- Train Epochs [✓ Train Error] Test Epochs Test Error
- Fitness Limit: 100

Figure 7-1 Genetic Parameters Dialog Box

The various fields have default values which are good starting points for genetic optimization. With the exception of Pool Size, we will leave them set to their defaults for this example, but you should review the description of the parameters in Section 8.2.7 and experiment with their functions. For now, change the Pool Size setting to 10. Confirm **OK** to accept the settings. This will result in a message asking you to confirm that you want to initialize the Genetic Supervisor state. Confirm **OK**. The Genetic Supervisor is now ready to run.

Select the **Run Genetic Supervisor** command from the **Neural** menu. This will cause a dialog box to appear with an entry field to limit the generation count.

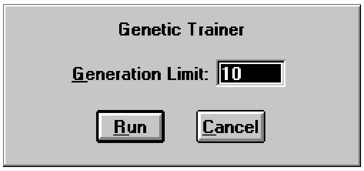
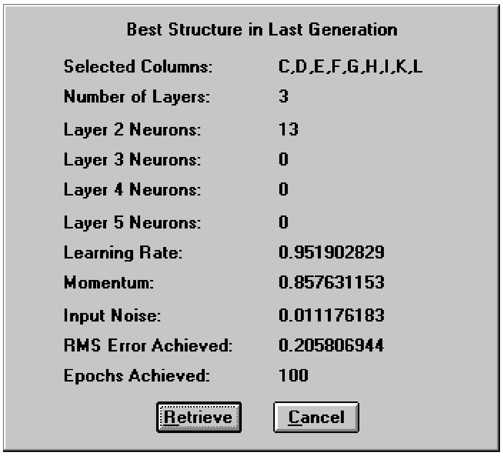


Figure 7-2 Genetic Trainer Dialog Box

The Genetic Supervisor will stop at the Generation Count which matches the setting. The default setting is Generation Count 10. Confirm **OK** to accept the settings. The Genetic Supervisor will now begin training and evaluating multiple neural network configurations to find the one which can provide the least RMS Error for a given number of training epochs. Note that this is only one evaluation condition that is possible, there are three others which emphasize the fewest training epochs for a given RMS Error and the same two conditions but applied to the testing set rather than the training set.

The Genetic Supervisor will report the Generation Count, the Structure Count, and the Least RMS Error or Least Epochs for the most recent completed generation. This will continue for several minutes until the Generation Count reaches the limit set in the **Run Genetic Supervisor** dialog box. At this time, the best structure identified in the current generation, after 10 generations of optimization, will be reported in a status display dialog box.



Best Structure in Last Generation

Selected Columns:	C,D,E,F,G,H,I,K,L
Number of Layers:	3
Layer 2 Neurons:	13
Layer 3 Neurons:	0
Layer 4 Neurons:	0
Layer 5 Neurons:	0
Learning Rate:	0.951902829
Momentum:	0.857631153
Input Noise:	0.011176183
RMS Error Achieved:	0.205806944
Epochs Achieved:	100

Figure 7-3 Best Structure Dialog Box

You have the option of retrieving the structure that is displayed or not doing so. If you chose not to retrieve the structure, you could resume the Genetic Supervisor at the stopped point by executing **Run Genetic Supervisor** again and increasing the generation limit to a higher Generation Count. This would proceed with additional generations of genetic optimization up to the new limit. At stopping points, you can also change the Pool Mode, Genetic Operators, or Fitness Criteria, without forcing a reinitialization of the Genetic Supervisor. However, changing most neural network configuration settings or the structure definitions in the Genetic Parameters dialog box would invalidate the optimization done to that point and require a reinitialization.

In this case, go ahead and confirm the retrieval with **Retrieve**. This will cause the structure that was shown to you to overwrite the Neuralyst Working Area with settings that duplicate the structure. The worksheet now contains a better optimized neural network. The settings which define the more optimal neural network are saved, but not the connection weights. So to try out the new neural network configuration, train and run the network as before by performing a **Train Network** command and then a **Run/Predict with Network**.

7.8 Operating Techniques

The Neuralyst Genetic Supervisor is a powerful tool, but as mentioned before, it can require lengthy optimization runs to achieve the best results. There are some techniques that are worth following when operating the Genetic Supervisor to get the most out of it.

When using a Fitness Criteria of Train Epochs or Test Epochs, which means that epoch count is optimized in reference to a set RMS Error level, not all neural network configurations will be able to achieve the set RMS Error level. These neural networks will continue training unless stopped by some other means. The external effect is that the Genetic Supervisor becomes “stuck” on a single structure. Remember that only the parameters specified in the Genetic Parameters dialog box are varied by the Genetic Supervisor, the remaining parameters that are settable in the Network Parameters and Enhanced Parameters dialog box are effective where they make sense. In particular, the three training cutoffs of Epoch Limit, Time Limit, and Error Limit are effective even under the Genetic Supervisor. Generally, it is best to set a Time Limit, with a value approximately one and a half to two times the amount of time that a trial run takes to achieve the set RMS level.

When using the Genetic Supervisor on a neural network model that has relatively few Input columns with a small Pool Size setting and an Inclusion Rate of notably less than 1, it is often possible for an initial generation to be created where no structure has all the Input columns specified, or where all Input columns are specified but the other structure characteristics cause the candidate solution to be culled. As a result, it can take many generations for cross-breeding or mutation to bring the Input column set back up to a full count for consideration. Under these conditions it is best to set Inclusion Rate to 1 or almost 1 and allow mutation to eliminate columns or to increase the Pool Size so the full Input column set is more likely to receive immediate evaluation.

When observing the Genetic Supervisor, it often happens the reported Least RMS Error or Least Epochs will increase in a current generation from the preceding one. These variations are not unusual and occur because the weight sets are initialized randomly. This may result in the best structures having slight variations in their reported results from generation to generation even though a single structure remains the best candidate solution.

Also, while observing the Genetic Supervisor, it often happens that the reported Least RMS Error or Least Epochs will attain an effective plateau after a few generations, subject to the minor variations mentioned previously. It is possible that this means that the Genetic Supervisor has attained an optimal solution. However, it also happens that after attaining a plateau for many generations, a sudden mutation can cause a marked improvement and a new, improved performance level. There is no clear way to predict the difference. This is very similar to biological evolution.

Generally, if the Genetic Supervisor has attained a plateau for a number of generations, the majority of structures will have become bred to be like the best structure. The propagation of the best structure in this fashion will generally take fewer generations than the set Pool Size if the Pool Mode is set to Closed Pool or Immigration. That is, if the Pool Size is set to 10, then the majority of structures will become similar in fewer than 10 generations. At this point, the Genetic Supervisor will attain better results only if Immigration is set or if there is a mutation. When this happens, it is possible to stop the Genetic Supervisor and change the settings of the Pool Mode, Genetic Operators or Fitness Criteria; then restart the Genetic Supervisor with the effect of stressing and evolving the population in a different way, with possibly beneficial results.

As a final suggestion, the Genetic Supervisor can deliver very useful results even with short runs. This can be achieved by setting the Genetic Supervisor to a large pool with only one or two iterations. This is an automated way to test a random space of neural network configurations for the best performer. If the neural network is moderately sized, the selected neural network will often be close to an optimal network.