Contents

Qnet V2.1 - 32-bit Neural Modeling for Windows

TABLE OF CONTENTS

Qnet Reference:

Example Problems:

Miscellaneous:

Introduction

Welcome to the world of neural networks. There is currently enormous interest in neural systems and what can be accomplished by employing them to solve problems. Qnet has been designed to provide both the expert and novice with a powerful tool for creating and implementing neural networks into everyday problem solving.

A neural network is best defined as a set of simple, highly interconnected processing elements that are capable of learning information presented to them. The foundation of neural network theory is based on studies of the biological activities of the brain. A neural network's ability to learn and process information classifies it as a form of artificial intelligence (AI).

The most exciting feature of this new technology is that it can be effectively applied to a vast array of problems, many of which have been thought to be too complex or lacking in sophisticated theoretical models. Neural networks are responsible for making significant advances in the traditional AI fields of speech and visual recognition. Investment managers are creating investment models to better manage money and improve profits. Scientists and engineers use them to model and predict complex phenomena. Marketing professionals are employing neural networks to accurately target products to potential customers. Geologists can increase their probability of finding oil. Lenders use neural networks to determine the credit risk of loan applicants. Complete neurocomputing hardware systems are being built for use in everything from automobiles to manufacturing systems. The variety of problems that can be solved effectively by neural networks is virtually endless.

Qnet is a neural modeling system that is designed to exploit today's more powerful PC hardware and operating systems. Qnet executes in 32-bit protected mode of 386, 486 or Pentium™ class CPU's. Network sizes are limited only by the computer system's memory and disk capacity. Qnet also takes advantage of significant performance gains offered by running in the CPU's protected mode -- up to 40% faster than standard 16-bit Windows applications. Qnet offers advanced network design features for creating complex networks that learn using highly optimized backpropagation training algorithms. Qnet features include:

- Blazing speed. Speeds over 700,000 connections per second on Pentium based PC's are possible with Qnet. Accuracy and stability are retained by Qnet with full 32-bit floating point representation of all training information (unlike integer based neural modeling software).

- On-line help. Help is available for all menus, dialog screens and input items.

- Fast, easy network design. Select transfer functions on a layer-by-layer basis to give networks unique characteristics. Qnet's connection editor can be used to customize network connections and control logic flow through the network.

- Easy data interfacing. Training data is easily imported to Qnet via the universally compatible ASCII file format with support for space, comma and tab delimited formats (these formats are supported by virtually all spreadsheets, word processors, text editors and database programs). Import data directly into Qnet's optional DataPro data preprocessor

- Automated test set inclusion for overtraining and model integrity analysis. This powerful feature takes the guess work out of determining when a network is properly trained and suitable for use. Test sets may be selected from the training data using a variety of methods, including randomly.

- Complete interactive analysis of the training process using NetGraph™ and its powerful AutoZoom feature. NetGraph instantly creates graphs of all key network and training information. AutoZoom can be used to interrogate plotted information to any level of detail required.

- Sophisticated network analysis tools. Interrogate your network to find relative importance of network

inputs. Analyze node contributions to network performance. Perform tolerance checks to quickly determine accuracy. All tools are on-line and can be accessed interactively during the training process.

• AutoSave to automatically store the network model during training. The AutoSave feature protects you from overtraining and network divergence by allowing you to retrieve stored network information from any prescribed point of the training run.

• Learn Rate Control to automate network training. Qnet takes the drudgery out of adjusting learn rates during training. The Learn Rate Control feature improves convergence times and promotes stable network training with minimal user interaction.

• Multiple training algorithms. Highly optimized backpropagation and FAST-Prop methods can be employed interchangeably during network training.

• A network recall mode. Recall mode can be used for analyzing trained networks with new sets of data. The same powerful network analysis features used for training are available in Qnet's recall mode.

• Source code for integrating trained networks into your own applications. Programmers will appreciate the ANSI C source code that allows you to easily interface any application with royalty-free neural models developed with Qnet. Or use our optional product QnetTool to automate the integration of your Qnet developed neural networks into your everyday Windows applications.

• Example problems. The example problems included with Qnet will get you started learning and using Qnet immediately.

All these features combine to make Qnet the most powerful and easy to use neural network model generator available. Regardless of your familiarity (or lack thereof) with neural networks, it is strongly recommended that you read through Qnet's help file or user's guide before using Qnet. Familiarity with Qnet's features and options will greatly improve your productivity with this software.

System Requirements

Qnet runs on compatible PC's with a minimum configuration of:

386, 486 or Pentium™ or higher CPU (486DX or Pentium+ recommended).
Windows 3.x with Win32s, Windows 95 or Windows NT.
VGA or higher graphics adapter (800x600 or higher recommended).
4 MBytes of system memory (8+ recommended).
Minimum 2 MBytes of free hard disk space for installation.
Mouse.

Math coprocessors are strongly recommended for Qnet. Training neural networks is a computationally intensive activity and central processors alone are simply not powerful enough to handle the rigors of controlling an advanced GUI and performing the required training computations. The good news is that floating point math coprocessors are now common in the PC marketplace and have become increasingly affordable. All 486DX, 486 OverDrive™ and Pentium™ class computers have built-in coprocessors. Any 386 class computer is easily upgradeable by adding an inexpensive 387 coprocessor. Intel offers OverDrive™ upgrades for 486SX™ computers that will add a math coprocessor and speed multiplying technology. We can assist you with any upgrade required for your system. Qnet also comes with versions specifically optimized for Pentium or 486 CPU's to get the most out of your hardware.

The system memory required for Qnet depends on the size of the neural models you expect to create. While Qnet has a virtual memory feature that allows you to run model sizes larger than the system memory capacity, training performance will begin to degrade dramatically if network sizes become significantly larger than system memory. For small to average sized problems, Qnet will require between 1 to 2 MBytes of memory. To adequately run Windows, Qnet and additional applications a minimum of 8 MBytes RAM is recommended. Your particular Windows installation may require more memory than that suggested here. Be aware that certain device drivers, disk caching utilities and other OS functions will reduce the amount of system memory available Qnet and other applications.

A mouse is required for certain functions with Qnet's graphing utility, NetGraph, and to productively navigate through Qnet's point-and-click menu system.

Installation

Qnet Installation:

NOTE: If you're installing to Windows 3.1 and Win32s is not installed on your system (or you are unsure) go to section 3.2 now.

Prior to installing Qnet, please terminate any active applications. This assures access to any required installation resources. Insert the Qnet's install ***disk 1*** in the appropriate drive for the media. Select ***FILE/RUN*** (or Start/Run) from the Program Manager (or Taskbar) to run the Qnet's *Setup* program. Type ***a:setup*** (or b:setup) in the command line field to start. Setup will request that you confirm both the "Install From" location and the "Install To" location. The default installation location for Qnet is ***C:\QNET21***. If you desire to have Qnet reside in a different location, alter the "Install To" location. Change disks as requested by the setup procedure.

Qnet will install the following files to the selected Qnet directory:

QNET.EXE	**Qnet for Win32**
QNET.HLP	**Qnet Help File**
DATAPRO.EXE	**DataPro application**
DATAPRO.HLP	**DataPro Help File**
QNETTOOL.EXE	**QnetTool application**
QNETTOOL.HLP	**QnetTool Help File**
OCR.XLS	**Sample Excel/QnetTool use**
OCR.WB1	**Sample Quattro Pro/QnetTool use**
FILELIST.TXT	**List of all files and proper installation locations**
***.NET**	**Example Qnet network definition files**
***.DAT**	**Data files for examples problems**
\SOURCE	**subdirectory containing C source file**

Additional resource files are installed to the Windows system directory. The file ***FILELIST.TXT*** has the complete list of installed files and gives the locations they would normally be installed to. If troubles occur during installation, refer to this file to verify if individual components are correctly installed *(NOTE: your particular Windows setup/configuration has control of the ultimate locations. Qnet's setup program queries your Windows installation for proper locations).*

Win32s Installation:

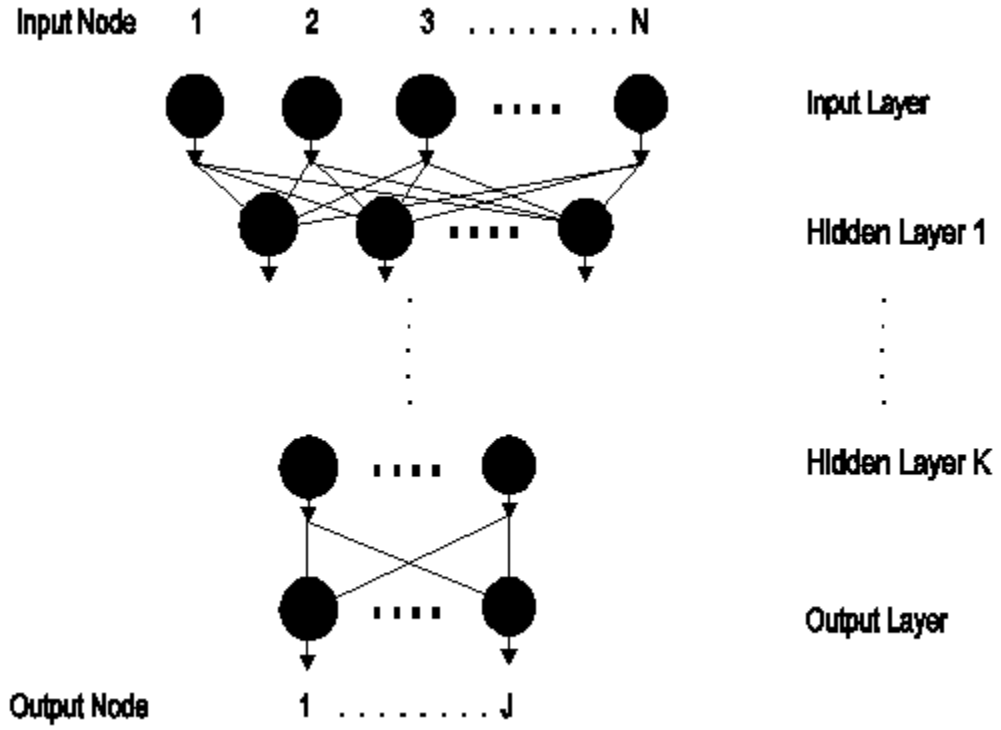
NOTE: Not required for Windows NT or Windows 95 systems.

If you are installing Qnet V2.1 with Windows 3.1, Microsoft's Win32s (version 1.2 included) upgrade must be available prior to installing Qnet. Insert ***Win32s 1*** into the appropriate drive to begin installation. Select ***FILE/RUN*** from the Program Manager to run the setup program (type **a:setup** - or b:setup - in the command line field). Follow Microsoft's installation instructions to properly install Win32s.

IMPORTANT: Win32s requires the DOS SHARE utility for proper file IO function. If Win32s instructs you to add this to your AUTOEXEC.BAT, do so prior to running/installing Qnet. Simply add SHARE to the end of your AUTOEXEC.BAT file. Neither Microsoft's Win32s setup nor Qnet's setup performs this function).

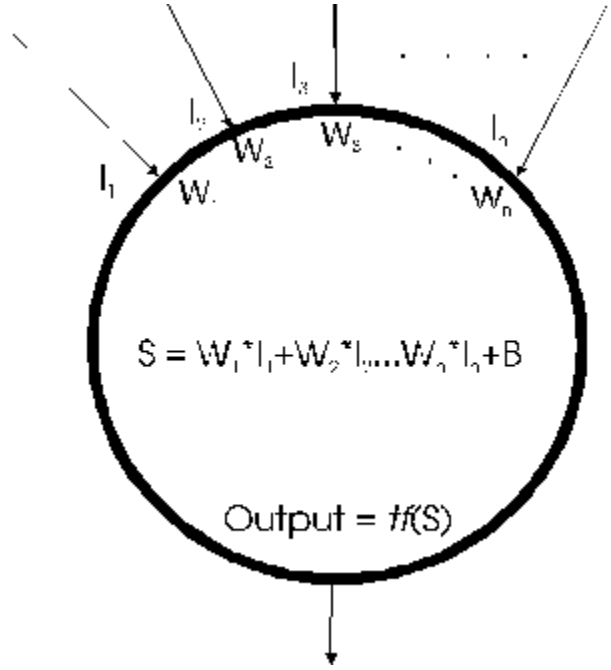
Neural Networks - A Brief Description

A human brain continually receives input signals from many sources and processes them to create the appropriate output response. Our brains have billions of neurons that interconnect to create elaborate "neural networks". These networks execute the millions of necessary functions needed to sustain normal life. For some years now, researchers have been developing models, both in hardware and software, that mimic a brain's cerebral activity in an effort to produce an ultimate form of artificial intelligence. Many theoretical models (termed paradigms), dating as far back as the 1950's, have been developed. Most have had limited real-world application potential, and thus, neural networks have remained in relative obscurity for decades. The backpropagation paradigm, however, is largely responsible for changing this trend. It is an extremely effective learning tool that can be applied to a wide variety of problems. Backpropagation related paradigms require supervised training. This means they must be taught using a set of training data where known solutions are supplied.



Backpropagation type neural networks process information in interconnecting processing elements (often termed neurons, units or nodes—we will use “nodes”). These nodes are organized into groups termed layers. There are three distinct types of layers in a backpropagation neural network: the input layer, the hidden layer(s) and the output layer. A network consists of one input layer, one or more hidden layers and one output layer. Connections exist between the nodes of adjacent layers to relay the output signals from one layer to the next. Fully connected networks occur when all nodes in each layer receive connections from all nodes in each preceding layer. Information enters a network through the nodes of the input layer. The input layer nodes are unique in that their sole purpose is to distribute the input information to the next processing layer (i.e., the first hidden layer). The hidden and output layer nodes process all incoming signals by applying factors to them (termed weights). Each layer also has an additional element called a bias node. Bias nodes simply output a bias signal to the nodes of the current layer. Qnet handles these bias nodes automatically. They do not need to be included or specified by the user. All inputs to a node are weighted, combined and then processed through a transfer function that controls the strength of the signal relayed through the node’s output connections. A nodes operation is shown in figure below. The transfer function serves to normalize a node’s output signal strength between 0 and 1. Qnet provides four transfer functions: the sigmoid (default), gaussian, hyperbolic tangent and hyperbolic secant functions.

Qnet allows the transfer function to be selected on a layer-by-layer basis to create unique hygrid networks. Network processing continues through each layer until the network's response is obtained at the output layer.



When a network is used in recall mode, processing ends at the output layer. During training, the network's response at the output layer is compared to a supplied set of known answers (training targets). The errors are determined and backpropagated though the network in an attempt to improve the network's response. The nodal weight factors are adjusted by amounts determined by the training algorithm. The iterative procedure of processing inputs through the network, determining the errors and backpropagating the errors through the network to adjust the weights constitutes the learning process. One training iteration is complete when all supplied training cases have been processed through the network. The training algorithms adjust the weights in an attempt to drive the network's response error to a minimum. Two factors are used to control the training algorithm's adjustment of the weights. They are the "learning rate coefficient", eta, and the "momentum factor", alpha. If the learning rate is too fast (i.e., eta is too large), network training can become unstable. If eta is too small, the network will learn at a very slow pace. The momentum factor has a smaller influence on learning speeds, but it can influence training stability and promote faster learning for most networks. Qnet uses a sophisticated control scheme that adjusts the learning rate coefficient to keep network training proceeding at a near optimal pace.

_

Network Design and Construction

When designing a network, the modeler must specify the following information:

- ***The number of input nodes.***
- ***The number of hidden layers (1 to 8).***
- ***The number of nodes in each of the hidden layers.***
- ***The number of output nodes.***

The following additional design features can also be set:

- ***The connection design of the network.***
- ***The transfer functions used in each layer.***

Qnet's network design dialog used to set or alter these network design options.

Input and Output Layers

The input layer of a neural network has the sole purpose of distributing input data values to the first hidden layer. The number of nodes in the input layer will be equal to the number of input data values in the model. For example, assume a lender wishes to create a neural network that will accept or reject automobile loan applications. Inputs could include things such as the loan applicant's age, marital status, the number of dependents, education status, total family income, the total monthly debt payments (house, cars, credit cards, etc.) and the monthly payment required for the new loan. If this is the extent of input information for the model, then this network would be designed with 7 input nodes. The output of this model is simply whether the person is qualified or not (1=yes, 0=no). Therefore, the output layer would consist of one node. Each case of 7 inputs and 1 output is referred to as one training pattern. If there is previous loan information for 5000 people, the model could be trained using 5000 patterns.

The number of nodes in the input and output layers are set in the network design dialog.

Hidden Processing Layers

In the loan application example it is clear that determining the number of input and output nodes is trivial once the data model has been formulated. Choosing the number of hidden layers and the number of hidden nodes in each layer is not so trivial. The construction of the hidden processing structure of the network is arbitrary. While there is normally a large envelope of hidden layer constructions that yield like results, the importance of selecting an adequate hidden structure should not be underestimated. Many factors play a part in determining what the optimal configuration should be. These factors include the quantity of training patterns, the number of input and output nodes and the relationships between the input and output data.. It may often be tempting to construct a network with many hidden layers and processing units -- falling into "the bigger the brain the better the model" trap. This philosophy can easily result in a poorly performing model. When a network's hidden processing structure is too large and complex for the model being developed, the network may tend to memorize input and output sets rather than learn relationships between them. Such a network may train well but test poorly when presented with inputs outside the training set. Also, network training time will significantly increase when a network is unnecessarily large and complex. The concept of memorization learning versus cognizant or generalized learning will be explained in detail in chapter 9. Generally, it is best to start with simple network designs that use relatively few hidden layers and processing nodes. If the degree of learning is not sufficient, or certain trends and relationships cannot be grasped, the network complexity can be increased in an attempt to improve learning. A plausible starting point for the loan application model would be to use 2 hidden layers with 3 to 4 nodes per layer. If this design does not train sufficiently, the size and complexity of the hidden structure can be increased. For this problem, memorization would not be likely due to the relatively large number of training patterns (5000).

It has been demonstrated theoretically that for a given network design with multiple hidden layers, there will always exist a design with a single hidden layer that will learn at an equivalent level. However, in practice, it is usually better to employ multiple hidden layers for solving complex problems. To adequately model a complex problem, a single hidden layer design may require a substantial increase in the number of hidden nodes compared to a 3, 4 or 5 hidden layer construction. In simple terms, a single hidden layer design with 10 nodes may not learn and perform as well as a network with two hidden layers containing 5 nodes each. Multi-hidden layer networks tend to grasp complex concepts more easily than networks with one layer. One reason for this is that the multi-hidden layer construction creates an increased cross-factoring of information and relationships. Thus, a network's learning ability is controlled by both the total number of hidden layers and the total number of hidden nodes.

Qnet allows up to 8 hidden layers (experience has shown that the vast majority of problems will work fine with 4 or less hidden layers). The number of nodes per layer in Qnet is limited only by the memory available and practical limits in processing speed. Expect the practical limits to be closely tied to your processor speed and memory capacities.

The design of the hidden processing structure is specified in the network design dialog

Network Connections

Another network design consideration concerns how to control the network's connections. Qnet implements a connection editor that allows connections to be removed from the fully connected default configuration. This allows logic flow to be introduced to the network. Input information can be channeled and processed in a localized area of the network. "Pass-thru" nodes can be constructed that receive only one input connection from the preceding layer and pass that information down to the next layer. This has the effect of creating connections that skip a layer. While the connection editor gives the modeler almost unlimited flexibility in designing a network, the fact is that the vast majority of designs work best fully connected. Qnet's connection editor is best suited for highly advanced models that require groups of input data to be processed through separate network pathways.

Connections are set in Qnet's network connection editor dialog and is accessed from the Training Setup/Network Design dialog.

Transfer Functions

A node's transfer functions serves the purpose of controlling the output signal strength for the node (except for the input layer which uses the inputs themselves). These functions set the output signal strength between 0. and 1. The input to the transfer function is the dot product of all the node's input signals and the node's weight vector. Qnet gives you the option of selecting two distinct types of transfer functions: the sigmoid and the gaussian. The functions are selectable on a layer-by-layer basis in Qnet and networks can be created that incorporate both types. Figures 6.1 and 6.2 show the behavior of each function.

This sigmoid function is Qnet's default transfer function and it is the most widely used function for backpropagation neural networks. The sigmoid function is represented by the mathematical relationship 1/(1+e-x). The sigmoid function acts like an output gate that can be opened (1) or closed (0). Since the function is continuous, it also possible for the gate to be partially opened (i.e. somewhere between 0 and 1). Models incorporating sigmoid transfer functions usually exhibit better generalization in the learning process and often yield more accurate models, but can also require longer training times.

The gaussian transfer function can greatly alter the dynamics of a neural network model. Where the sigmoid function acts like a gate (opened, closed or somewhere in-between) for a node's output response, the gaussian function acts more like a probabilistic output controller. Like the sigmoid function, the output response is normalized between 0 and 1, but the gaussian transfer function is more likely to produce the "in-between state". It would be far less likely, for example, for the node's output gate to be fully opened (i.e. an output of 1). Given a set of inputs to a node, the output will normally be some type of partial response. That is the output gate will open up partially. Gaussian based networks tend to learn quicker than sigmoid counterparts, but also tend to produce networks that are prone to memorization with less generalized learning.

The hyperbolic function counterparts to the sigmoid and gaussian functions are the hyperbolic tangent and hyperbolic secant functions. The hyperbolic tangent is similar to the sigmoid but can have exhibit different learning dynamics during training. It can accelerate learning for some models, but it also may not achieve the same accuracy as a sigmoid based models. Experimenting with different transfer functions with your particular models is the only way to conclusively determine if any of the non-sigmoid transfer functions will improve learning characteristics.

For the vast majority of network designs, we suggest that the sigmoid function be used as the transfer function. A general rule of thumb (but not always the case), is that the sigmoid will produce the most accurate model and, likely, the slowest learning. If you intend to frequently train similar models and/or training speeds will be important, you may wish to experiment with different combinations of transfer functions, including hybrid networks, in search of the fastest training models that produce acceptable accuracy.

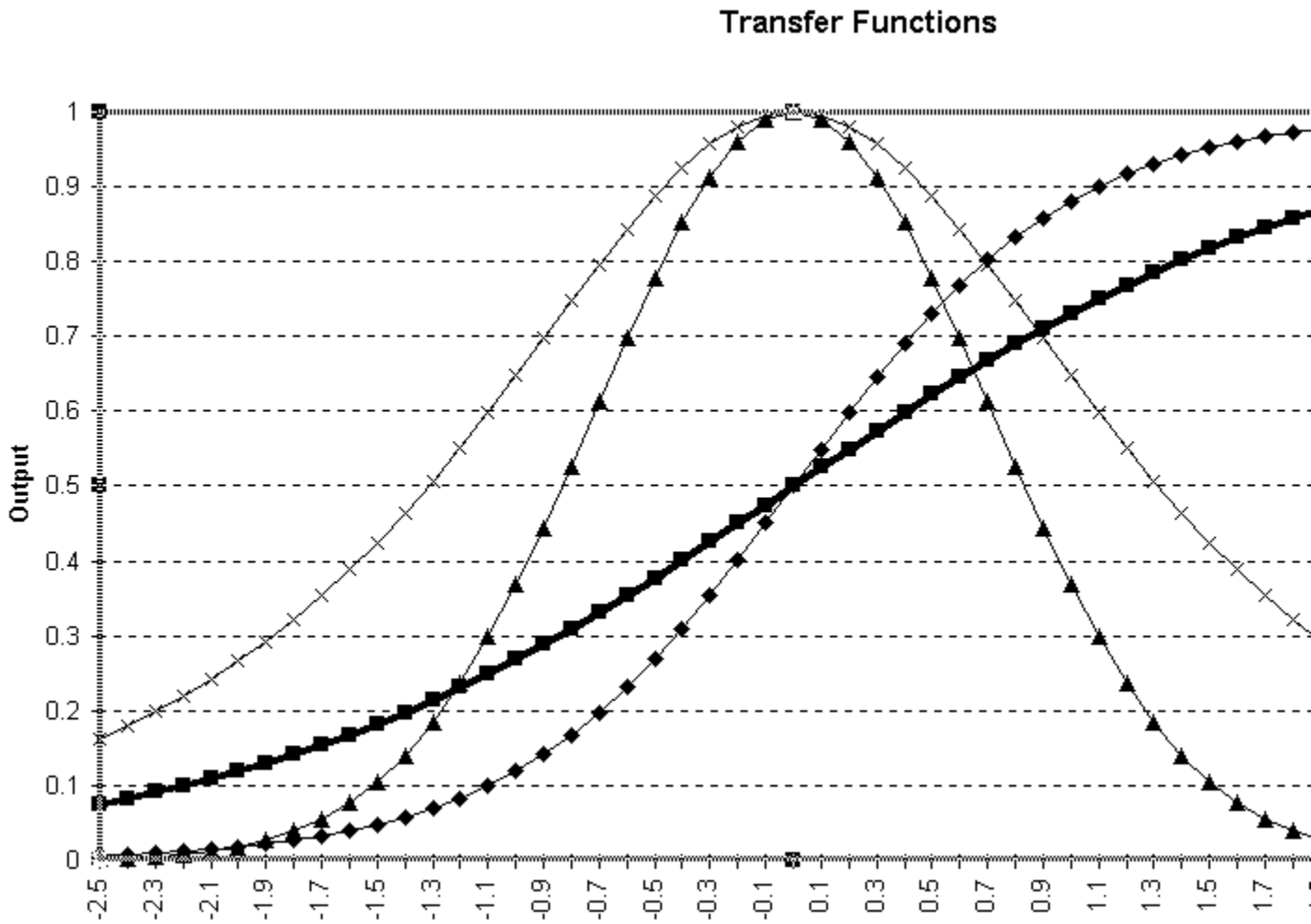


Figure 6.1 - Sigmoid, Gaussian, Hyperbolic tangent and secant transfer functions normalize the output signal generated by each node.

Training Data

Before a network can be created and trained by Qnet, data for the model must be organized and formatted for compatibility with Qnet. The files containing the training (or recall) data are specified in the training (or recall) setup dialog windows. The steps required to create training data for Qnet involve:

Gathering the training cases.
Determining what, if any, data preprocessing should take place.
Formatting the final training set for Qnet.

With backpropagation neural networks, the more training data that is gathered for the training process, the better the model will likely be. With more training cases available, the modeler is able to consider increasingly complex network designs. Gathering a large number of training cases will also make it easier to employ rigorous test sets for overtraining analysis and model integrity checks. Once the model information is gathered or generated, data preparation and formatting are required. These tasks are easily accomplished by using any of today's popular spreadsheet or database programs. Once compiled, data may be saved to a compatible ASCII file format or directly transferred to Qnet's DataPro for training.

Input File Format

Preparing training data for use with Qnet is an easy process. Training data files use the universally compatible ASCII (text) row/column (columnar) input format. Each data column represents data for one input node or a target for one output node. Each row in the file represents one input training case or pattern. Using the loan application example, the data columns would include the age, marital status, number of dependents, years of education, total family income, total monthly debt payments, and the monthly payment required for the loan. There is also one output node, the qualification status. The target data for the output node(s) can be located in the same file as the inputs or in a separate file. If the same file were used for both, then the loan application training file would consist of 8 data columns. If the historical database contained 5000 loans, the training file would have 5000 rows (i.e., lines or records).

Data column delimiters can be any combination of spaces, commas or tabs. The only requirement is that both the input node data and output node targets be in contiguous data columns. For the loan example, the data columns containing the input node information can be 1 through 7, 2-8, 10-16, etc. You simply tell Qnet which data column to start reading from. The same rules apply for the target data used by the output node(s). Blank and commented lines are ignored. Comments can be inserted in the files by starting the line with a "#" character. The following is a template of what an input file may look like:

```
# THIS IS A COMMENT
<INPUT NODE 1>  <INPUT NODE 2>  <INPUT NODE 3>  <INPUT NODE 4> ..... <OUTPUT NODE 1> ..... <=== PATTERN 1
<INPUT NODE 1>  <INPUT NODE 2>  <INPUT NODE 3>  <INPUT NODE 4> ..... <OUTPUT NODE 1> ..... <=== PATTERN 2
<INPUT NODE 1>  <INPUT NODE 2>  <INPUT NODE 3>  <INPUT NODE 4> ..... <OUTPUT NODE 1> ..... <=== PATTERN 3
<INPUT NODE 1>  <INPUT NODE 2>  <INPUT NODE 3>  <INPUT NODE 4> ..... <OUTPUT NODE 1> ..... <=== PATTERN 4
                                                .
                                                .
                                                .
                                                .
```

It is not required that the input node data columns precede the output node data columns if the two sets of information are contained in the same file.

Qnet also supports the use of column labels. If you choose to use column labels, they should appear in the first record (or line) of the file and start with the comment ("#") character. Labels, if present, will be used when plotting and viewing training and recall data. The format for labels should be as follows:

#"Label 1" "Label 2" "Label 3" ....

As with data, the labels can be delimited by commas, spaces or tabs. Each label must be enclosed in quotes.

The use of a spreadsheet as a training data preprocessor to Qnet is highly recommended. A spreadsheet will allow you to group columns, move, add or eliminate rows and perform virtually any type of data preparation required. If the training data can be imported into or has been generated with a spreadsheet application, it is very easy to format and save the data in an ASCII (text) format. For example, Microsoft Excel™ allows any spreadsheet to be saved in a formatted text mode with comma (***.CSV***), tab (***.TXT***) or space (***.PRN***) column delimiters — all compatible with Qnet. Quattro Pro for Windows™ allows data to be saved to ***.TXT*** text files that are tab delimited. Refer to your spreadsheet documentation for further information. If the loan application problem were arranged in an Excel spreadsheet, the setup could appear as follows:

#Age	Married (1Y,0N)	Dependents	Education (0,1,2,3)	Total Income	Monthly Debt	Loan Payment	Qualified?
24	**0**	**1**	**0**	**$26,000**	**$732**	**$399**	**0**
35	**1**	**3**	**3**	**$66,000**	**$1412**	**$299**	**1**
58	**1**	**6**	**2**	**$120,000**	**$3800**	**$2200**	**0**
44	**0**	**1**	**1**	**$39,000**	**$500**	**$300**	**1**

(Education code - 0 No HS, 1 HS, 2 College, 3 Higher)

The "**#**" character in front of the "Age" label is used to create a Qnet comment record. Also, for many spreadsheets, you will need to turn off special formatting features like *currency* or *percent* type formats (the currency format is used in the income, debt and payments columns above). Spreadsheets are often designed to write the "***$***" or "***%***" signs into the output file. ***These type of number formats are not supported in Qnet.*** The number formats supported by Qnet are integer (non-decimal numbers), floating point (numbers with decimals) and scientific notation (numbers with exponential formats). Also, make sure that negative numbers are preceded with a "***-***" sign and not enclosed with parentheses and that commas are not used in partitioning large numbers (i.e. 1,000,000). In our example, after performing the necessary reformatting operations (removing currency and currency/comma formats), a ***.CSV*** (Comma Separated Values) file could be written for use with Qnet. If the file is saved as ***LOAN.CSV***, this information would be set in Qnet's Training Setup/Training Data dialog window. The starting column locations of input node data, column 1, and the target data, column 8, must also be specified.

Likewise, all popular database applications can create data files in an ASCII text format. If the training data is coming from your own private application, simply follow the above rules when writing to formatted text files. The prepared data file is specified in Qnet's Training Setup/ Training Data Specification.

An alternative method to creating an ASCII file from your spreadsheet, is to transfer your training data directly into Qnet's DataPro application and save the data to a Qnet compatible file using DataPro. See DataPro for more information on this method.

For very large models with hundreds or thousands of input and output nodes, each line in the file can become quite long. Qnet's has no limit on the length of each line or record that will be scanned for data. Most spreadsheet programs limit output to 256 data columns. When generating a Qnet file that would contain data for 1000 input nodes, several files would have to be combined to create the entire input set. If you require utilities for working with large data sets, contact Vesta for assistance.

Data Preparation

Proper data preparation can make the difference between successful and unsuccessful neural models. Some models will benefit greatly from simple transformations of the input and target data. For this reason, it is important to understand how different training data representations will influence the neural model being created.

Neural network training data falls into two classes: continuous valued and binary. For many inputs the data can be processed and represented in either of these formats. Let's assume we wish to create a model that will project the monthly sales of widgets and is going to include the month of the year as one of the inputs. We can either represent months as a continuous value from 1 to 12 through a single input node or as 12 separate nodes using binary inputs. For the binary case, all nodes would be set to 0 except for the month we wish to project sales for. As a second example we wish to predict the direction of a stock's value. Should we predict the following day's value of the stock as the actual price, a percentage change from the current week's level or as a simple binary value indicating up or down? Clearly, decisions must be made. Making the right choice can make or break the model being designed.

When deciding between continuous values or a binary representation, one must consider the impact upon what is being modeled. For the widget example, assigning continuous values of 1 through 12 to represent the month implies a predetermined ranking for each month. For many models, we would have no reason to believe that August should be better than April or that January should be less than November. The sale of widgets may have distinct monthly patterns that have nothing to do with the month's chronological order. Creating 12 binary input nodes avoids the implied ranking problem. The neural network that uses one input node may produce acceptable results, but a considerable amount of extra training will be required to decode the implied ranking.

The optical character recognition problem included with Qnet provides another example of binary data representation. The model's output is a determination of what number (0 through 9) has been presented to the network through a bitmap picture. We could construct a model with one output node corresponding to the actual number recognized, or we could set up 10 output nodes with each node representing one digit. Using 10 output nodes is the proper way to formulate this model. This is because the process of recognizing a character is independent of the character's numerical value. Forcing the network to assign a numerical ranking to the output will unnecessarily complicate the main task of recognizing the character. The ten output node design will simply output a 1 to the appropriate node when a number has been recognized. In practice, when new and somewhat different images (or fonts) are presented to the network for recognition, we may not get an exact 0 or 1 reading from the output node. The optical character recognition program utilizing the neural network may require that the output of a node be greater than some threshold (say .5) before it will consider a character recognized. If multiple nodes indicate some degree of recognition, the one with the greatest output strength would likely be selected.

For the stock forecasting example, continuous values would provide more information than a binary representation indicating simply whether the market is up or down. Knowing the magnitude of the up or down change will improve model learning by providing additional information. Also, the size of the up or down prediction will likely correlate with the probability of the model predicting the right direction.

It is also important to consider how a continuous value should be represented. A major pitfall the neural modeler must avoid is the use of unbounded inputs or targets. If one were to choose the stock's price as the target value, substantial problems would result. The stock's future value has no upper bound. Once the value moves outside the historical trading range or the stock splits, the model will become obsolete or perform poorly. This problem can be eliminated by using a percentage change format. Excluding highly volatile swings, one can be confident that the percent change will fluctuate within a range of around ±5% for most days. This provides a reasonable upper and lower bound for the target values. If isolated, volatile swings produce a few weekly changes significantly outside the typical range, consider limiting those changes to some maximum limit (like 5%). This will prevent isolated cases from unduly impacting network predictions and the data normalization process.

Another data preparation problem can occur when there is an extremely large amount of input node data

to model. This problem is common with backpropagation neural networks used for visual recognition. Take a case where a neural model is to be created to monitor the quality of a weld on a production part going down an assembly line. A camera will provide the neural model with a picture of the weld, and the part will be accepted or rejected based on this picture. The input to the neural model will be a digitized picture from the camera. The output of the network will be to simply accept or reject the part (binary). If the digitized picture has a resolution of 1000x1000, the total number of input nodes is 1 million (i.e., one node per pixel). While Qnet can theoretically handle a problem this large, computer speed and memory limitations will likely prevent a network of this size from being trained and put into practical use. The solution is to compress the video information in some manner to reduce the total amount of information that must be modeled. A simple way to reduce the image size is to tile the image. This involves averaging neighboring pixels to reduce the overall quantity of inputs. A second method is to use Fast Fourier Transforms to compress the image into a series of waveforms. The waveform coefficients are used as network inputs instead of the actual pixel data. This technique has been used quite successfully with visual recognition problems.

Data Normalization

Backpropagation neural networks require that all training targets be normalized between 0 and 1 for training. This is because an output node's signal is restricted to a 0 to 1 range. Qnet also requires input normalization to improve training characteristics. Qnet can perform the data normalization automatically and keeps the normalization details from the user. If this option is selected during training setup, all data for the nodes in the input layer and/or training targets for the output layer will be normalized between the limits of .15 and .85. If new training patterns are added to the training set in subsequent sessions, the data will be re-normalized as necessary. (Note: This may make the network weights slightly out of sync with the training data. A small amount of training will be required to recover.) Even if the training data is already between the limits 0 and 1, normalization may still be desirable. For example, if all target data is between .01 and .02, it would be better to normalize the data over a wider range so that the network can better resolve and predict the targets.

When Qnet's automatic normalization is used, the entire normalization process becomes invisible to the user. All network inputs and outputs will be returned to their original scale when plotted, printed or saved to a file. For recall mode, input node data will be normalized to the same limits used for network training. Network outputs will be automatically scaled to the proper range if automatic normalization was selected for the training targets. If new input data is being presented to the network, there is always a chance that the new normalized data will fall outside the 0 to 1 limits. This may not be a problem, however, it should be noted that when inputs to a network are significantly different from the data ranges that were used during training, the model's accuracy must be questioned.

In the loan application example, Qnet's automatic normalization would be used for the network inputs (the output is in a binary form and does not require normalization). If we assume that the number of dependents ranged from 1 to 10 for the training data and a loan applicant later applies with 18 dependents (this is only an example), could the network accurately predict whether the person should qualify for the loan? The model will produce an answer, but the result may be of questionable accuracy.

Training Patterns - Quantity

The number of cases that are used for training is extremely important. For backpropagation neural networks, the more training patterns that are used, the better the resulting model will be. The only limit with Qnet is the speed and memory of the computer system and the ability of the modeler to collect training cases. A large number of training cases allows the network to generalize and learn relationships easier. Also stated previously, the size and complexity of the network structure can be increased, since training set memorization becomes less likely. Training sets with a small number of patterns run the risk of being memorized, even by small network configurations.

Qnet also allows the use of ***test sets*** during training. Test sets are patterns set aside from the training set to test for network overtraining and to check the integrity of the model. If a model cannot respond intelligently to patterns outside the training set, then the model will be of little value (unless training set memorization is the goal). Qnet allows the user to indicate the number of patterns to allocate for the test set and then to monitor the network's test set response error during training. The test patterns should be contained in the same file(s) as the training patterns. You can instruct Qnet to select a certain number of records randomly or a specified number of patterns can be used from the beginning or ending of the training file(s). If the patterns in the training data file(s) are in some type of systematic order, it is best to select the test patterns on a random basis so that test set is not a limited subset of the training cases. Normally, the training set will contain many more patterns than the test set. Ratios of training patterns to test patterns are commonly in the range of 5 or 10 to 1 (and higher).

Training Neural Networks with Qnet

Once the training data has been organized for the model and the network design features have been chosen, network training can begin. At this point Qnet should be started, a new network definition file selected and a training run initiated. The training setup dialog window is used to specify the network configuration, the training data file information and the initial training parameters. Depending on model size and complexity, the training process may take minutes, hours or even days for very large problems. Qnet's fast execution speeds and assorted analysis tools are designed to simplify the training process. The graphing and analysis tools make it easy to determine how well a network has learned and when a network has been trained to its optimal level of performance. The training parameters may be interactively changed during the training process. Important training topics include:

Learning Modes
Learning Rates and Learn Rate Control
Patterns Processed per Weight Update Cycle
Starting New Networks
Training Error
Training Analysis Tools
Unattended Training
Training Divergence
AutoSave
Backpropagation vs. FAST-Prop
When is Training Complete?
Training Speed

Learning Modes

An important concept to understand about the training process is that there are two distinct types of learning that can take place. One type is generalized learning where the network develops an understanding of how the inputs can best be generalized to formulate an output prediction. The other type is "memorization" where the network can, in effect, recall a set of outputs after being presented with the inputs. An example of generalized learning is where a person studies how to add together a small set of numbers (i.e., 2+2, 3+5, etc.) and through understanding some basic concepts, that person can determine the results of any two numbers presented to him (even if they were not previously studied). An example of memorization learning is where a person learns the US state capitals. Learning the capitals of 45 states does little to help a person predict what the other 5 might be. Memorization learning is only useful for the learned set. It offers no help in determining solutions outside the learned set. The sample problem "RANDOM" included with Qnet shows an example of memorization learning. While memorization does have some benefits for certain models, cognizant learning is desired for the vast majority of real-world neural models. Differentiating between the learning modes will allow you to optimize model training and better determine the effectiveness of the a model prior to practical use.

Qnet's training algorithms attempt to drive the network's response error for the training set to a minimum value. The error value monitored during Qnet training is the root-mean-square (RMS) error between the network's output response and the training targets (equivalent to the standard deviation). When the training set's error is descending during the training process, one or both types of learning discussed above is taking place. Unfortunately, there is no way to determine which type of learning is taking place by monitoring the training set error by itself. To determine the type of learning, a test set (or overtraining set) must be employed. Qnet allows the test set to be monitored interactively during training. This set of data is not used to train the network, however, the error in the network response is monitored to determine how the network responds to patterns outside the training set. If both the training and test set errors are declining, cognizant learning predominates since the network is learning to generalize the relationships between the inputs and outputs. If the test set error increases while the training set error declines, then memorization is the predominant learning mode. When a test set's error has reached a minimum level and begins to increase indefinitely thereafter, overtraining is occurring. Overtraining a network after this minimum has been reached can actually hurt the predictive capabilities of the model being developed.

It should be noted that the method of determining the learning modes and overtraining status by monitoring the training and test set errors assumes that the test set is an adequate subset of the training set. This may not always be true. For problems where the test set is some limited or organized subset of the training set cases, the minimum test set error may simply indicate the point that the network has best modeled that subset of test set cases. To function as a true overtraining indicator, it is important that the test set cases be a truly random and broad sample of the training cases. To help prevent this problem, Qnet may be instructed to select the cases randomly from the training set. While this does not guarantee a perfect set for testing, a high probability will exist that an ample amount of unique case types will be present when large test sets are employed.

Learning Rates and Learn Rate Control

The backpropagation training paradigm uses two controllable factors that affect the algorithm's rate of learning. To optimize the rate at which a network learns, these factors must be adjusted properly during the training process. The two factors are the learning rate coefficient, eta, and the momentum factor, alpha. The valid range for both eta and alpha is between 0 and 1. Higher values adjust node weights in greater increments, increasing the rate at which the network attempts to converge, while lower values decrease the rate of learning. Just as there are limits to how fast a brain can learn ideas and concepts, there are also limits to the rate at which a network can learn. If a network is forced to learn at a rate that is too fast, instabilities develop that can lead to a complete divergence of the training process.

The learn rate coefficient can be controlled manually during training or Qnet can control it automatically using its Learn Rate Control (**LRC**) feature. LRC will drive eta higher or lower in a systematic fashion depending on the current learning activity. If the network appears to be learning at a relatively slow rate, eta is driven up quickly. Conversely, if the network is learning at a fast pace, Qnet will raise eta only slightly, hold it constant, or even lower it to avoid instabilities. If at any time the network shows signs of instability (seen as oscillations in the training error), eta is lowered quickly to damp the instabilities. Damping instabilities is critical to preventing complete training divergence. The LRC feature can be turned on and off interactively during the training process, and it can be activated at setup time by specifying the iteration number that LRC will start.

Occasional interaction with the LRC system can help to improve the learning process. For example, let's assume that a network is training with LRC active. After several hundred iterations NetGraph is used to view the eta history. The graph shows that whenever eta exceeds a value of 0.15, instabilities occur (seen as oscillations in the RMS error) and eta is dropped substantially to avoid divergence. During each recovery process, learning slows due to lower learning rates. By setting appropriate minimum and maximum values for eta, LRC will keep eta below the established upper limit and above the specified minimum. For the above example, eta could be limited to a maximum of 0.12 to prevent the instabilities from occurring. This will keep the network learning at an optimal pace by preventing the slow downs required to recover from instabilities. It is common for these limits to change gradually over many iterations (usually the upper limit decreases during the training process, but not always). Repeat this procedure when instabilities develop on a regular basis.

LRC concerns itself only with control of eta. Usually, little or no interaction is required with the momentum factor. The momentum factor damps high frequency weight changes and helps with overall algorithm stability, while still promoting fast learning. For the majority of networks, alpha can be set in the 0.8 to 0.9 range and left there. However, there is no definitive rule regarding alpha. Some networks may train better with alpha values set at a lower level. Some networks train perfectly with no alpha term used at all (set to 0). Most neural modelers prefer to use higher momentum values, since this usually has a positive effect on training. If training problems occur with a given alpha value, it may be helpful to experiment with different values. Alpha can be changed interactively at any time during the training process with Qnet.

Note: For the vast majority of networks, LRC is an effective tool preventing divergence and keeping eta in a range that will improve learning speeds. If a model exhibits poor learning characteristics with LRC active (i.e. training divergence or many instabilities), simply turn LRC off (Options menu of the training window) and set eta to a value low enough to guarantee stable learning. Networks employing gaussian transfer functions may find that LRC is less effective in preventing divergence in some cases.

When all training cases are not used in each weight update cycle (Patterns per Weight Update Cycle), LRC is not recommended. Error descent can be somewhat noisy and/or training characteristics can be adversely affected by varying the learn rate. Taking manual control of the learning rate for such models is recommended.

Starting New Network Training

A new network is initialized by setting all processing node weights to random values. Qnet does this automatically for new networks. This option can also be selected for any network during setup or training to reset the network to an untrained state. When initiating the training process for a new network, several decisions must be made regarding certain training parameters.

First, a seed value for the learning rate must be chosen during training setup. A learn rate value in the 0.005 to 0.1 range usually works well for new networks. If the initial guess turns out to be too high and the network diverges (see section 9.7), simply reset the network by selecting "Options/Initialize-Reset Weights" from the training window's menu bar. Select "Options/Set Learn Rate" to try a lower learning rate value.

A second consideration for new networks concerns the iteration number at which LRC should be activated. When training begins with a new network, the training error can oscillate wildly. This is normal behavior for new networks. If Qnet's LRC option is active during these initial oscillations, eta will be lowered in an attempt to eliminate them. This can slow training by driving eta to an artificially low value. To prevent this from occurring, set the "Learn Rate Control Start Iteration" item in the training parameters setup dialog window to at least 50 or 100 iterations. The LRC option can also be turned on and off interactively during the training process.

If a long training session is planned, the number of iterations should be set to a very large number. The number of iterations can be changed during the training session or training can be interactively terminated at any time.

Other Qnet training parameters can normally be kept at their default values. The parameters include: the FAST-Prop Coefficient, the minimum and maximum learn rate settings for LRC, the momentum factor - alpha, the patterns per weight update, the screen update rate, and the AutoSave rate. These parameters can be adjusted during the training process if required.

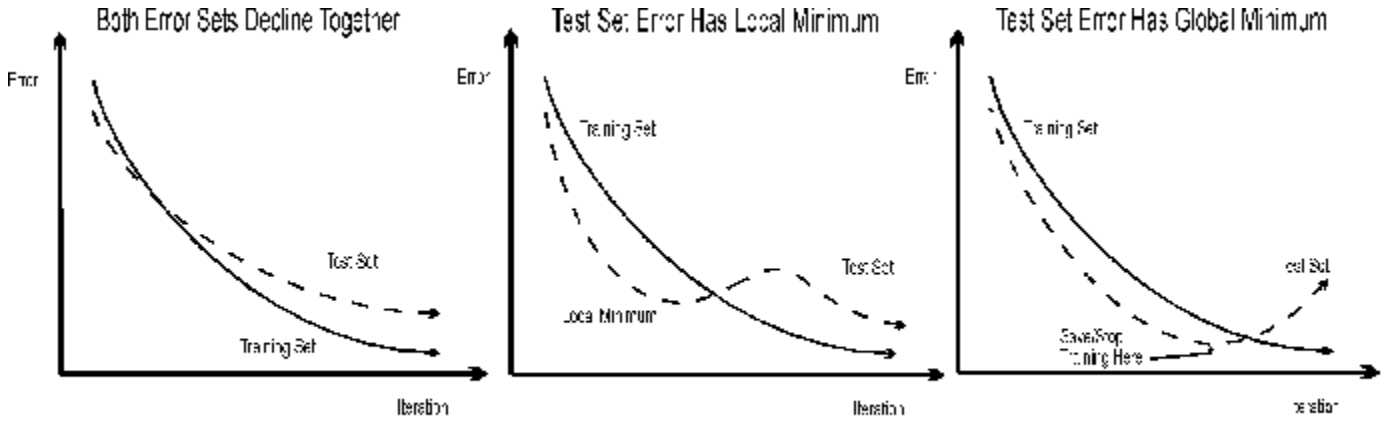
Training Error

Monitoring the error histories is the quickest way to determine the training progress of a network. Along with the visual readout of both training and test set errors, complete error histories for the run can be obtained with NetGraph. This provides the modeler with the most information about the progress and the relative state of a network's convergence. NetGraph's AutoZoom feature can be particularly helpful when viewing error histories. AutoZoom allows the modeler to zoom in on features of interest that might be obscured by the scaling used for the initial plot.

By monitoring the training set's RMS error, the modeler can determine the pace of network learning, the frequency of instabilities and the general state of convergence (or divergence). Qnet's training algorithms attempt to drive the training set error to a minimum value. As a network reaches this converged or steady state, the error value will approach some minimum value.

By monitoring the test set's RMS error, the modeler can determine the overtraining status of the network and how well the network responds to cases not contained in the training set. For some networks, the test set's error will simply decline along with the training error to some minimum value. Another possibility is that it will decline, reach a minimum, and then increase indefinitely thereafter, even though the training set's error continues to decrease. For this case, the model should only be trained to the point of the test set's minimum error. When the test set error begins to increase it can be assumed that memorization is predominating and overtraining has begun. Unfortunately, determining this point is not that simple. False or local minimums may occur in the test set error. These local minimums indicate that some mix of the learning methods is taking place. Some networks may exhibit long periods of training where the test set error increases before declining again. Figure 9.1 depicts these possible scenarios. (Note: To minimize the effect of test set error computations on training speed, Qnet computes the error value during screen update iterations only. Setting large screen update intervals will limit your ability to monitor the test set error.)

If a test set's error begins to increase, training should be continued to determine whether the minimum is local or global in nature. The AutoSave feature of Qnet allows you to return to a point at or near the minimum if it is global in nature. AutoSave will store the network at selected intervals during training. This interval can be specified during setup or training. To retrieve an AutoSave network, select "File/Save AutoSave File..." from the training menu. You will be asked to specify a network file name and an iteration value (from a list of iterations at which the network was saved). Select the iteration that is nearest, but still prior to the start of overtraining. If additional training is required from this iteration, exit the current training session and open the newly saved network file for training.



When either the training set error or test set error begins to increase while using the FAST-Prop training algorithm, return to standard backpropagation by setting the FAST-Prop coefficient to zero. While the

FAST-Prop method of training can accelerate the learning process, this training method can at times get "stuck". The training and/or test set errors may start to increase or fluctuate. Standard backpropagation does not experience this problem.

NOTE: When all training cases are not used in each weight update cycle (Patterns per Weight Update Cycle), error values may exhibit relatively noisy behavior during convergence. Also, LRC should not be used in this case either. Manual control of the learning rate is recommended.

Training Analysis Tools

Many tools are available for analyzing the training process. Tools are available for interrogating the quality of agreement with training and test targets and for looking at key model development issues. Available NetGraph tools include:

RMS Error History plot: The training error history can be monitored to determine the rate of network learning and it can be used to determine when learning has reached its maximum level. Other interesting information can be derived from the training and test set error history plots. It is common to find long "plateaus" in the error level where no significant learning takes place. This behavior is particularly common when multiple hidden layers are being employed. This indicates that the network is trying to "figure out" certain input/output relationships. Plateaus are often followed by steep descents in the training error, yielding accelerated periods of learning. It is important that "plateau" conditions are not mistaken for a converged network. Another common feature in error history plots are minor oscillations representing training instabilities. Training instabilities are quickly damped by the LRC feature, if active. When oscillations occur frequently, consider augmenting LRC by setting a maximum eta that LRC should not exceed.

Correlation History plot: The correlation coefficient measures how well the network predictions trend with the targets in the training set. The range of the correlation coefficient is from -1 to 1. The closer the coefficient is to 1, the more accurate the predictions. The closer to 0 (or below), the less accurate and more random the predictions become. This plot often trends opposite the RMS error, the correlation increases as RMS error decreases. It can, however, be more informative because it uses an absolute scale to better quantifies the agreement (1 is perfect linear correlation, 0 is random). The extreme targets and predictions are the most heavily weighted in the calculation of the correlation coefficient (for binary output types all cases are at extremes).

Tolerance History plot: The tolerance history produces the percentage of training set predictions that fall within the user's pre-defined tolerance of the targets. This can be beneficial when a pre-defined accuracy must be achieved by the model. The history also helps to better quantify when additional training is not producing tangible or measurable improvements in the training set agreement.

Test RMS Error plot: This plot is used for overtraining analysis and helps determine how well the network generalizes learned information. This plot depicts how well the network predicts cases not used in the training process.

Test Correlation History plot: The correlation coefficient measures how well the network predictions trend with the targets for the cases outside the training set. The correlation range is between -1 to 1. As with the the RMS error of the test set, the correlation history can be used to determine overtraining. While the correlation and RMS errors often trend together, they can give slightly different indications on the onset of overtraining.

Test Tolerance History plot: The tolerance history produces the percentage of test set predictions that fall within the user's pre-defined tolerance of the targets As with the test set RMS error and the correlation coefficient, this percentage can also be used to help determine where generalized learning is optimal and overtraining has begun.

Learning Rate History plot: The learn rate (eta) history can be viewed to determine how Qnet's Learn Rate Control has adjusted eta during training. Use this option to determine what eta limits should be applied to Qnet's LRC option to help avoid training instabilities.

Targets/Network Outputs plots: Qnet provides three separate plot formats for viewing training targets and network outputs. The quickest overview plot is the "Targets vs. Network Outputs" plot. This plot displays network predictions vs. the targets for all output nodes on a single plot (all data remains normalized so that all output nodes will share a common scale). The closer the points fall on a the plotted "X=Y" line, the better the overall agreement for the model. Training and test set points are plotted with different symbols. Network predictions and targets may also be compared separately for each output node. In this case, "Targets/Net Outputs vs. Pattern Sequence", the information is plotted versus the

pattern sequence number. Up to three separate curves will be shown: the training targets, the training set network responses and the test set network responses. The test and training set predictions can be distinguished by different colored curves or symbols. This plot format offers the best detailed view of the agreement between output predictions and the training targets. A final plot format shows the error between the network predictions and targets plotted versus the training pattern sequence number (separate plot for each output node).

Input Node plots: The input node data is displayed versus the training pattern sequence number (separate plot for each input node). This plot format can be used to scan the input node sets for possible data anomalies. It is recommended that input node plots be reviewed at some point during training to scan the inputs for bad data.

Input Interrogator: After a network is near its fully trained state, it is often useful to determine what inputs are important to a network's output response. The Input Interrogator will plot, for each output node, the relative importance of each input on that particular output. Sensitivities are determined by cycling each input for all training patterns and computing the effect on the network's output response. This plot helps to determine what the key inputs for the model are and which are not effective in formulating output predictions. Please note, this sensitivity study assumes that each input value is independent of all other inputs. For models where this is not true, some caution should be used when interpreting the results.

Input Color Contours: Choose any two inputs to visualize their contribution to formulating any output. Full color contours are produced depicting the influence of the inputs on the selected output.

Node Analyzer: The Node Analyzer plot helps determine how the hidden nodes are being utilized by the network. For networks that are over designed in the hidden layer structure, many nodes may contribute little or nothing to the output response. For each hidden layer in the network a plot will be generated comparing the relative strengths of all output connections for that layer. The plot shows the nodes' percent contribution to that layer's output signals over all training patterns. If there are many nodes in a layer that are showing limited contributions, then that layer may be specified with too many nodes. Likewise, if all nodes show strong contributions, it is possible the adding extra nodes will help the model.

Qnet also has an information viewer to further analyze network training and performance. In addition to viewing network node weights and output predictions and targets, the following information can be obtained:

Network Information: Detailed network information of a model's construction and training state can be viewed and printed. This information is particularly useful if you are sorting through several network designs. Prior to terminating training, printing this record will provide details that will be pertinent in comparing the current model with other model constructions. Having a detailed record for each model attempted will offer a quick way of comparing the results between models.

Statistics: The statistics option is used to compare network predictions with training targets. For each output node and for both training and test sets, standard deviations, biases, maximum errors and the correlation coefficient is computed and displayed. The standard deviation between the predictions and targets assumes a gaussian distribution exists in the prediction error. The bias value measures any shift between predictions and targets. This indicates whether the prediction is systematically high or low. If the error in the predictions is a true gaussian distribution, than the bias value should approach 0. The correlation coefficient is a statistical measure of how well the predictions agree with the targets. A value of 1 indicates perfect correlation. Values close to 0 (or below for this analysis) indicate that little or no correlation exists between network predictions and targets. As with all statistical comparisons, validity of these numbers increase with the greater number of cases used in the computations.

Tolerance Checking: The tolerance checking option is useful in determining the number of network predictions that fall within a selected tolerance from the training targets. This option will count the number of points that fall within the tolerance and total them in the "Correct" column. Points falling outside the tolerance are totaled in the "Wrong" column. Separate computations are made for both training and test

sets. If the goal is to achieve a network that predicts within a certain tolerance of the correct answer, this tool provides information on whether that goal has been achieved. Once the desired tolerance is successfully reached, network training can be terminated.

Threshold Checking: The threshold checking option is beneficial for analyzing network models with a certain type of output characteristic. Networks that are modeling an output that is essentially an up or down prediction (for example, a financial model that is predicting percent gains and losses), can use this option to gain insight into the quality of the network model. A threshold value is specified and only network predictions that exceed the threshold (+ or -) are counted "Correct" or "Wrong". "Correct" and "Wrong" is determined by the direction change of the actual target. If both the prediction and target move in the same direction (i.e. have the same sign) and the prediction exceeds the threshold, a "Correct" case would be counted. "Wrong" cases occur when the predicted direction is not correct. If the network prediction does not exceed the threshold, the case is ignored. By analyzing results of several threshold values, one can determine the point that the network model begins to yield reliable predictions. For models that do not have this type of output format, this option will not provide useful information.

Divergence Check: If a training divergence occurs during training, the diverged output node will produce all 0's or 1's for sigmoid transfer functions at the output layer and all 0's for gaussian transfer functions (normalized responses). This check scans the network output nodes for this behavior. The first output node found exhibiting this characteristic will be displayed. That particular output should be plotted to verify the diverged condition.

Qnet's rich set of real-time analysis tools allow the user to perform thorough and detailed model analysis. All tools are menu selectable during training making them accessible at any time. Using these tools during the training process will improve your ability to train and identify good network models.

Unattended Training

Large, complex models may require extended periods of unattended training. Overnight training must often be considered so that the PC will not be tied up during prime use hours. Several steps should be taken when planning unattended training. These include:

1) Make sure the "Time Remaining" field is at least as long as the period you wish to train for. If not, increase the number of iterations (Options/Iterations...).
2) Keep LRC on to guard against network divergence or set the learn rate to a reasonably low value.
4) Set the FAST-Prop coefficient to 0 (use standard backpropagation).
5) Set the Auto-Save rate to a reasonable level.

The AutoSave rate should be set so that if the run results in overtraining, you can return the network to its optimal training point in a reasonable length of time. For example, if it takes 10 minutes to perform 100 training iterations, setting the AutoSave Rate to 100 will guarantee that you can return to any training point within 10 minutes. Another important consideration when using AutoSave during long unattended sessions is disk space. Adequate disk space must be available to store the network model at the requested rate.

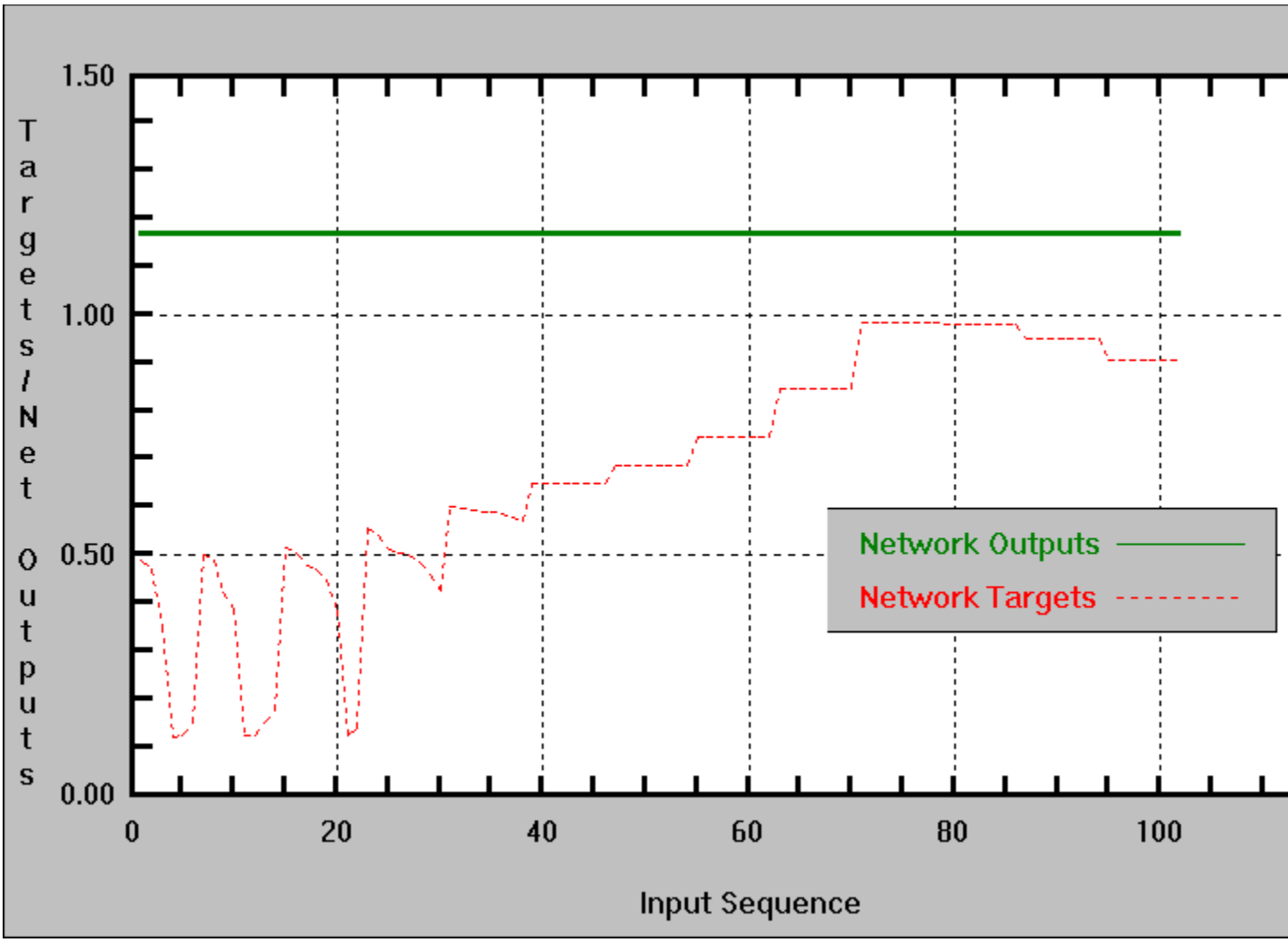
To safeguard your system's video display against screen burn-in during long periods of unattended training, we recommend that you use screen blanking or turning your monitor's power button off. ***DO NOT USE*** the animated screen saver utilities available under Windows. Animated screen savers will drastically reduce the CPU time available to Qnet. Your monitor's on/off switch remains the most efficient screen saver.

Training Divergence

During the training process, the network's learning pace may at times become too fast. When this happens, learning instabilities develop. These instabilities show up as small oscillations in the training error. If the learning rate is not lowered in response to the instabilities, network divergence can result. Qnet's Learn Rate Control helps prevent divergence by automatically lowering eta. If a network does diverge, the training RMS error will normally reach a large constant value (usually around 0.5). For multiple output node networks, it is possible to have a divergence at one output node and have continued convergence for others. In these cases, only a minor increase in the training error may be observed, followed by a continued decrease in the training error due to the continued convergence at other output nodes. Such cases are often difficult to detect by just monitoring network error.

Qnet offers two tools for detecting diverged conditions. From the training menu, selecting "Info/Divergence Checking" will quickly scan all nodes for diverged characteristics. The first output node that appears diverged will be indicated. It is also possible to spot diverged nodes by plotting each output responses with NetGraph (NetGraph/Targets-Outputs vs Pattern Sequence). If an output node's response is constant and completely outside the range of the target data, then that node has diverged. This indicates that the normalized response of the network is all 1's or 0's.

A diverged network must be either reset to a randomized state (select "Options/Initialize-Reset Weights" from the training menu) or returned to an iteration prior to divergence (using the AutoSave feature). Use the AutoSave recovery procedure for cases where a great deal of training would be lost by simply resetting the network. Setting a lower learn rate (eta) value or limiting LRC with a lower maximum eta will help to prevent the divergence from reoccurring.


1.50
1.00
0.50
0.00
Targets/Net Outputs
0
20
40
60
80
100
Input Sequence
Network Outputs
Network Targets

AutoSave

The AutoSave feature of Qnet allows you to easily recover from overtraining situations and training divergence. The user specifies a rate (or interval) at which the network should be stored. Qnet will store the network state in a temporary file at the selected interval during the training process. An AutoSave rate of 100 will cause the network state to be stored every 100th training iteration. The network state is not saved to your current network definition file. This must be done manually by selecting the File/Save Network option from the training menu. A network's state can be retrieved from the temporary AutoSave file and stored to a permanent network definition file by selecting File/Save AutoSave Network... When selected, you will be prompted for a network definition file name and the iteration number that is to be used in the recovery process. You may use the name of the current network definition file or a new one. The iteration number can be determined by viewing the training or test set error history. For overtraining, return to the first stored iteration prior to the point that the test set error started to increase. To recover from a diverged network, return to the first iteration prior to the divergence. The current training run is unaffected by this recovery process. If you wish to start training the network retrieved from AutoSave, exit the current training session and start a training run using the specified network definition file.

The temporary AutoSave file is destroyed when the current training run is terminated. You must retrieve any desired network states before exiting. Also, if you save a network state to the same file name as the one being used for the current run, do not save the current run prior to exiting or the retrieved network will be lost.

When selecting the AutoSave rate, several considerations must be made. Storing the network too often can slow execution speeds and increase disk space demands. Selecting very long intervals between AutoSave stores will mean that it could take considerable training time to get back to the desired network state. Limiting AutoSave about 15 minutes between stores will have the following benefits:

1) You can recover from a training problem in a reasonable period of time. For example, if overtraining started to occur at iteration 4800 and you had an AutoSave rate of 500 (assume 500 iterations per 15 minutes), you could reach the optimal training point in about 9 minutes by retrieving the network state at iteration 4500 and performing an additional 300 training iterations on that network file.

2) Longer time intervals between AutoSave will reduce the amount of disk space required. Small and medium size networks will require less than 100 KBytes of disk space for an overnight run if the saves are at least 15 minutes apart. If very long unattended sessions are planned (i.e., weekends or longer), caution must be used in selecting the AutoSave rate. A simple calculation determines the space required:

(bytes of disk space required) = (byte size of network's .NET file) * (# training iterations per minute) * (minutes of unattended training planned) / (AutoSave rate).

The "iterations per minute" number can be computed by dividing the "Max Iterations" by the "Time Remaining" field (converted to minutes) displayed at the start of any training run.

3) Using longer intervals between stores will reduce the impact of AutoSave on training speed. Disk writes are slow and can significantly retard execution times if they are performed often.

Backpropagation vs. FAST-Prop

The FAST-Prop coefficient controls the algorithm used by Qnet for training. FAST-Prop training can accelerate training for some networks and training runs can switch between FAST-Prop and backprop methods "on the fly" during training. If the FAST-Prop coefficient is set to 0 (the default), Qnet will employ its backpropagation algorithm to train the network. If the coefficient is set to a value above 0.0 (to a maximum of 3.0), the FAST-Prop algorithm is used. The closer the coefficient is set to 0.0, the closer FAST-Prop approximates standard backpropagation. While the FAST-Prop training method can often accelerate the learning process, a drawback with this method is that there is a risk that this algorithm will not converge to a minimum error, especially when higher coefficient values are used. For this reason, it is recommended that the training algorithm be switched to the standard backpropagation method at some point during the training process. Likewise, the FAST-Prop algorithm is **NOT** recommended for long periods of unattended training. Whenever FAST-Prop is being used, the training and test set RMS errors should be monitored closely for signs that the network is no longer converging. If either of these error values begin to increase, it is recommended that the FAST-Prop coefficient be set to 0. See Technical Overview for more information

When is Training Complete?

Determining when training has completed is not always a simple process. Often we'll know training was complete only after many additional iterations have been performed. For example, if a network is overtraining based on RMS error, we must check to make sure that the minimum test set error is not a local one. If the training error is making little or no progress in the downward direction, the network could be temporarily stalled at a learning "plateau".

Also, for some models, other factors may overide the standard RMS error analysis. There may be models that are better optimized using correlation or tolerance numbers. Overtraining in the test set may not occur in RMS error, correlation and tolerance at the exact same location. For some models, it may be more desirable to optimize training/overtraining on correlation or tolerance. You may wish to terminate training when any one of the three test set error tracking items begin overtraining or only after all three of the items reverse and overtrain. The methods to determine when a network has reached its converged or optimal state are:

1) **RMS error analysis:** Terminate the training process at the point where the test set error is at a minimum. The network is optimized for accuracy on cases available outside the training set. All cases carry equal weight.

2) **Correlation analysis:** Terminate the training process at the point where the correlation error is at a maximum. The network is optimized for minimum error by the test set cases when the correlation coefficient is at its maximum. This differs from RMS error in that the outputs and targets are weighted by their distance from the mean, thus, increasing the importance cases at the extreme minimums and maximums. Overtraining in this item occurs when the test set correlation coefficient declines indefinitely.

3) **Tolerance analysis:** If there is a known accuracy that needs to be reached, Qnet's tolerance checking can be used to determine when the network has reached the required level of accuracy. This item measures the percentage of cases where the prediction falls within a desired tolerance. Overtraining in this item occurs when the test set percentage declines indefinitely.

4) No further significant decline in the **training error**. If overtraining does not occur (or no test set is used) and a pre-defined tolerance does not exist, this method will optimize network accuracy for cases contained in the training set.

The appropriate method varies depending on the type of model being developed. Financial forecasting models may produce the best results using method 2 to optimize the extreme predictions. These are the cases that most influence the correlation coefficient and most likely will result in monetary transactions (buy/sell decisions). A minimum RMS error in the test set optimizes training for all cases equally. For all types of models, large, randomly selected test sets will produce the most accurate training/overtraining analysis and increase the likelihood that all methods will produce similar results.

Training Speed

Training speed is a critical factor for neural network software. Qnet's training algorithms have been highly optimized for speed in an effort to minimize training time. Training times are largely determined by your problem size and processor speed. There are several steps that can be taken to significantly improve your execution speed and limit convergence times. These include:

1) The first 4 Qnet iterations for any run will take noticeably longer than iterations there after. This is due to Qnet activating menu items as they become available during a run's startup. Base your training parameter tuning estimates like the screen update rate or AutoSave rate based on training speed after these initial iterations.

2) Pay attention to the rate of screen updating during training runs. During training, key network parameters are updated to the screen. How often this happens is controlled by the screen update rate. A value of 1 updates the screen each training iteration, a value of 2 updates the screen every second iteration and so on. Each screen update takes CPU cycles away from the solver. While this may seem insignificant, it can slow execution by 50% or more in extreme cases. To ensure that screen updating is not significantly retarding execution speed, limit screen updates to once every 2 or 3 seconds.

3) Use AutoSave rates that will yield several minutes between stores (10 to 15 recommended). Disk writes are extremely slow and can significantly retard execution times. By allowing several minutes between network stores, you will limit the effect of this feature on Qnet's performance.

4) Augment Qnet's LRC (Learn Rate Control) system to optimize convergence speed. Whenever instabilities become frequent, limit the maximum learning rate that LRC can use. Limiting eta in this way will improve convergence time by eliminating the overhead required to safely damp and recover from instabilities. Repeat this process whenever instabilities begin to occur regularly.

5) Do not use animated screen savers! These will compete with Qnet for CPU time and significantly slow execution during unattended training. The best screen saver is the power button on your monitor.

6) Keep Qnet the foreground task for unattended training. If your version of Windows allocates priority bases on whether the task is foreground or background, keep Qnet the foreground (or active) task to maximize execution priority.

Following these simple guidelines can greatly improve model development times. Other hardware issues can also greatly affect training times. If you continually build models that require Qnet to access virtual memory from disk, consider relatively inexpensive memory upgrades. If you intend to build and train many neural models consider CPU upgrades that will significantly improve your system speed.

Network Recall

Neural network recall is the processing of new inputs through a trained network. Qnet offers a variety of methods for accessing Qnet trained networks. These include the recall mode available within Qnet, C source for programmers to access trained networks in application development and Qnet's optional companion, QnetTool, that allows integration of trained networks into Windows based spreadsheet and data analysis applications.

To access your neural network from within Qnet, simply open your network definition file supply the file containing the new input node data in the recall setup dialog. Use the same file format rules that are required for training data. If targets are provided in recall mode (optional), the predictive qualities of a model can be checked. After all input patterns are processed, the network's output may be analyzed using NetGraph and Qnet's information viewer. Network outputs can also be saved to an ASCII file with tab delimited data columns for importing into spreadsheet applications for further analysis.

Source Code for Network Recall

The source file **QNETSOLV.C** can be used to incorporate trained networks into your own C or C++ applications. Only three functions need to be called to perform network recall:

1. NETDEF *init_net(char *NetFileName);
2. void net_output(NETDEF *net, float *inputs, float *outputs);
3. void free_net(NETDEF *net);

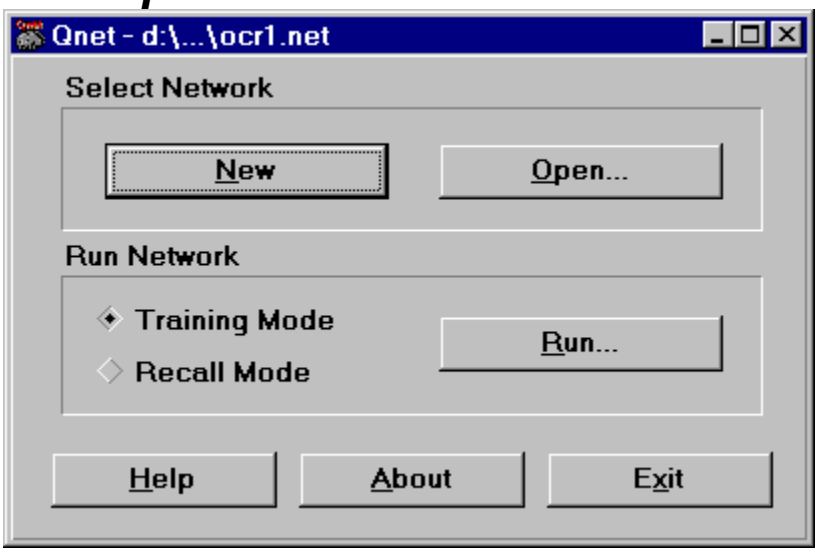
The routine, **init_net**, is used to initialize and allocate the neural network defined in the Qnet network definition file passed through the argument list. The routine returns a pointer to the C structure that defines the network. The user does not need to allocate or access members of this structure, simply define a pointer to **NETDEF** and **init_net** allocates and initializes the needed information. The routine **net_output** computes the network output. Arguments passed are the pointer to **NETDEF,** a pointer to a float array that contains 1 sequence of input patterns and a pointer to a float array that will contain the network outputs upon return from **net_output**. The order of values in the input array must be the same order that was used to train the network. At this point, you may write, save, or do whatever with the outputs. If you need to call **net_output** multiple times, simply loop through the desired number of calls, changing the "inputs" array before each call. The **free_net** routine frees all memory allocated for the network and should be called when all input patterns have been processed through the network.

A template of how a trained network is incorporated into sample C code is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include "qnetsolv.h"

//number of input nodes for network
#define NINPUTS 10
//number of output nodes for network
#define NOUTPUTS 2
main()
{
      float inputs[NINPUTS], outputs[NOUTPUTS];
      NETDEF *network;

// INITIALIZE NETWORK
      network = init_net("C:\\QNET\\TRAINED.NET");

// If multiple input cases for recall, add loop here!

// Read/set input values into float inputs[] array. The number
// of inputs must be equal to the number of input nodes
// and they must be in same order that was used for
// network training.

// COMPUTE OUTPUTS
      net_output( network, inputs, outputs );

// Write or use outputs. The number of output values is
// equal to the number of output nodes. The order is the
// same as the target data used for training.

// End Loop for multiple inputs here!

// FREE NETWORK STORAGE
```

```
        free_net(network);
}
```

Startup Window



The main window is the launching point for both Qnet training and recall runs. Main window options include:

New
Create a new, untitled Qnet network definition file and procede to training setup.

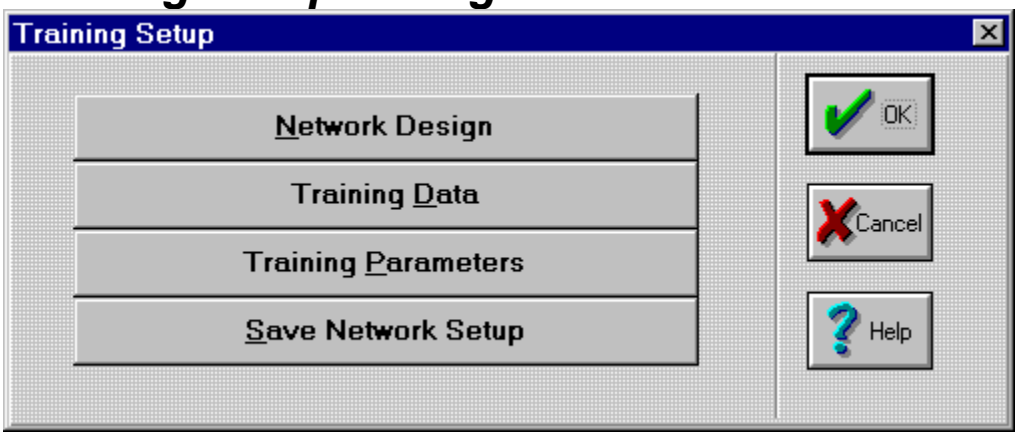
Open
Select an existing Qnet network definition file for training or recall operation.

Run
(Training Mode Selected)
Select this item to enter network training mode. Use this option to train both new and existing networks.
(Recall Mode Selected)
Select this item to enter network recall mode. Use this option to pass new sets of inputs through existing (trained) networks.

Help
Display this help file.

Exit
Exit Qnet.

Training Setup Dialog Window

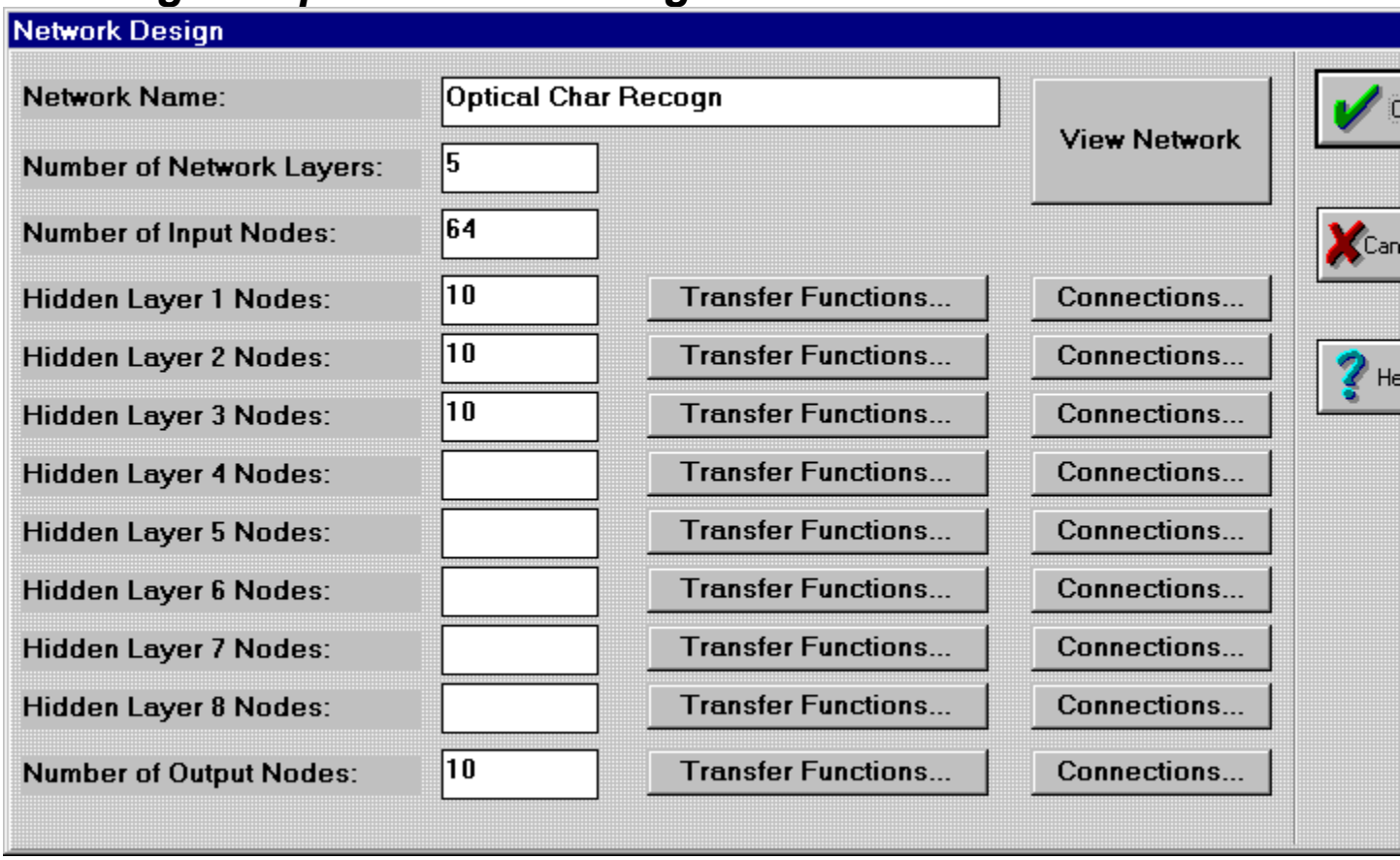


The training setup dialog window allows you configure and/or alter your neural network model. Options include:

Network Design
Training Data
Training Parameters
Save Network

All inputs are set to default values for new networks and must be modified to represent the model you are creating. For existing networks, values are initialized to those used in the previous run and may be modified as desired. A modified network may be saved to the same network definition file (overwriting the previous version) or as new file.

Training Setup - Network Design



Problem Name
Enter an identifying name for the network.

Number of Network Layers
Enter the number of layers for the network. Include the input layer, the arbitrary number of hidden layers and the output layer. The minimum value is 3 and the maximum value is 10 (i.e., 1-8 hidden layers). The following points should be considered when deciding on the number of hidden layers to use:

1) With more hidden layers, complex input/output relationships can be better modeled by the network with fewer total hidden nodes.

2) More hidden layers tend to slow training by increasing the total number of iterations required to learn. However, time per iteration can normally be decreased with the use of less total hidden nodes.
Using 3 to 6 total layers (1 to 4 hidden layers) is sufficient for the vast majority of neural models.

Number of Input Nodes
Enter the number of input nodes for the network. The number of nodes must correspond to the number of inputs in the network model. For example, if 10 inputs are used to model 3 outputs, the number of input nodes would be 10.

Number of Output Nodes
Enter the number of output nodes for the network. The number of nodes must correspond to the number of outputs in the network model. For example, if 10 inputs are used to model 3 outputs, the number of output nodes would be 3.

Number of Hidden Nodes per Layer
Enter the number of hidden nodes for each hidden layer of the network. The number of entries should be ***[NUMBER OF LAYERS - 2]***. The order is from the first hidden layer after the input layer to the last hidden layer before the output layer. Network designs tend to be better when the number of hidden nodes are matched to the size of the problem being modeled. The following points should be considered when choosing the number of hidden nodes in each hidden layer:

1) Fewer nodes are needed per each hidden layer when more hidden layers are used.

2) Highly complex the relationships between the inputs and outputs will require a greater number of hidden nodes per hidden layer to model those relationships.

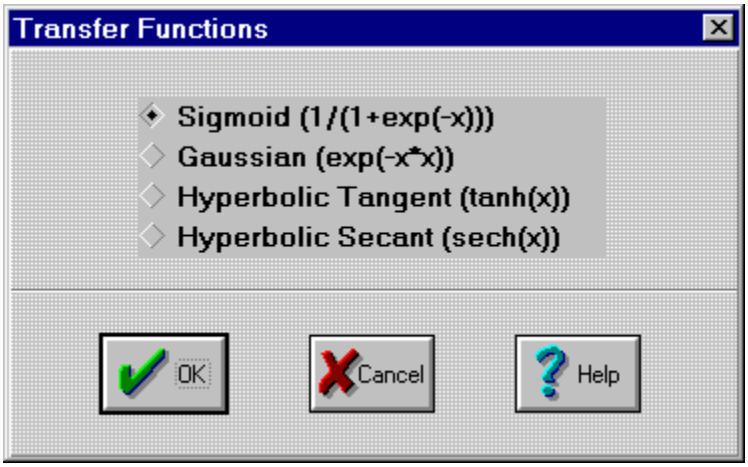
3) A larger number of hidden nodes may be used when there are a large number of training cases. When a small number of training cases are used, compared to the network size, memorization can become a problem.

4) When gaussian or secant transfer functions are employed, a greater number of hidden nodes usually improves model performance.

5) When using multiple hidden layers, specifying a constant number of nodes per layer or specifying a design that gradually decreases the number of hidden nodes in successive layers will generally produce the best results.

Specifying too many hidden nodes can result in poor models tend to memorize the training set rather than learn relationships. Specifying too few hidden nodes will result in models that can't learn the training data adequately. Some experimentation with the network construction may be required to determine the configuration that offers the best learning characteristics. Often, a large envelope of similar network constructions exist that will produce very similar results. For new models, it is best to start with smaller, simpler designs to validate the model prior to optimizing the network size. Qnet's node analyzer plot can be useful in determining when layers are over or under specified.

Transfer Functions: Select the transfer function for the hidden or output layers.



Sigmoid
Selecting the sigmoid button for a given layer assigns the sigmoid transfer function to all nodes in that particular network layer. A sigmoid transfer function is the default backpropagation transfer function used by each node in the network (except input nodes). The sigmoid function is represented by the mathematical relationship 1/(1+e^-x). It serves to normalize a node's output response to a value between 0 and 1. The sigmoid function acts like an output gate that can be opened (1) or closed (0). Since the function is continuous, it also possible for the gate to be partially opened (i.e. somewhere between 0 and 1). In general, networks using the sigmoid transfer functions learn and generalize relationships well and

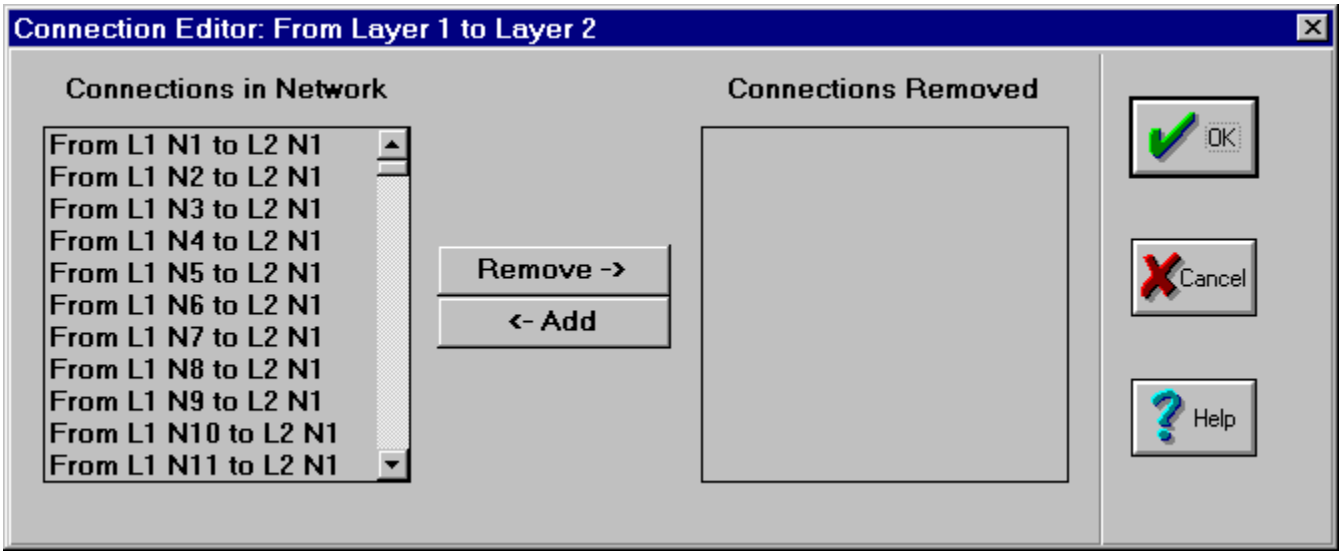
tend to produce more accurate models.

Gaussian
Selecting the gaussian button for a given layer assigns the gaussian transfer function to all nodes in that particular network layer. The gaussian transfer function can be used to change the learning characteristics of a backpropagation network. Where the default sigmoid function acts like a gate (opened, closed or somewhere in-between) for a node's output response, the gaussian function acts more like a probabilistic output controller. Like the sigmoid function, the output response is normalized between 0 and 1, but the gaussian transfer function is more likely to produce the "in-between state". It would be far less likely, for example, for the node's output gate to be fully opened (i.e. an output of 1). Given a set of inputs to a node, the output will normally be some type of partial response. Gaussian based networks tend to learn quicker for many models, but also may produce networks that are prone to memorization. Experimenting with different options, including hybrid networks that combine both types of transfer functions, can yield neural models that exhibit unique performance characteristics.

Tanh
The hyperbolic tangent is similar to the Sigmoid in both numerical and conceptual function. However, it is possible for hyperbolic tangent based layers/models to exhibit unique learning and performance characteristics.

Sech
The hyperbolic secant is similar to the Gaussian in both numerical and conceptual function. However, it is possible for hyperbolic secant based layers/models to exhibit unique learning and performance characteristics.
Experimenting with different options, including hybrid networks that combine multiple types of transfer functions, can yield neural models that exhibit unique performance characteristics.

Connections
The "***Connections...***" buttons invoke the connection editor for the given layer. Use this option to alter the network connections from the default fully connected configuration.



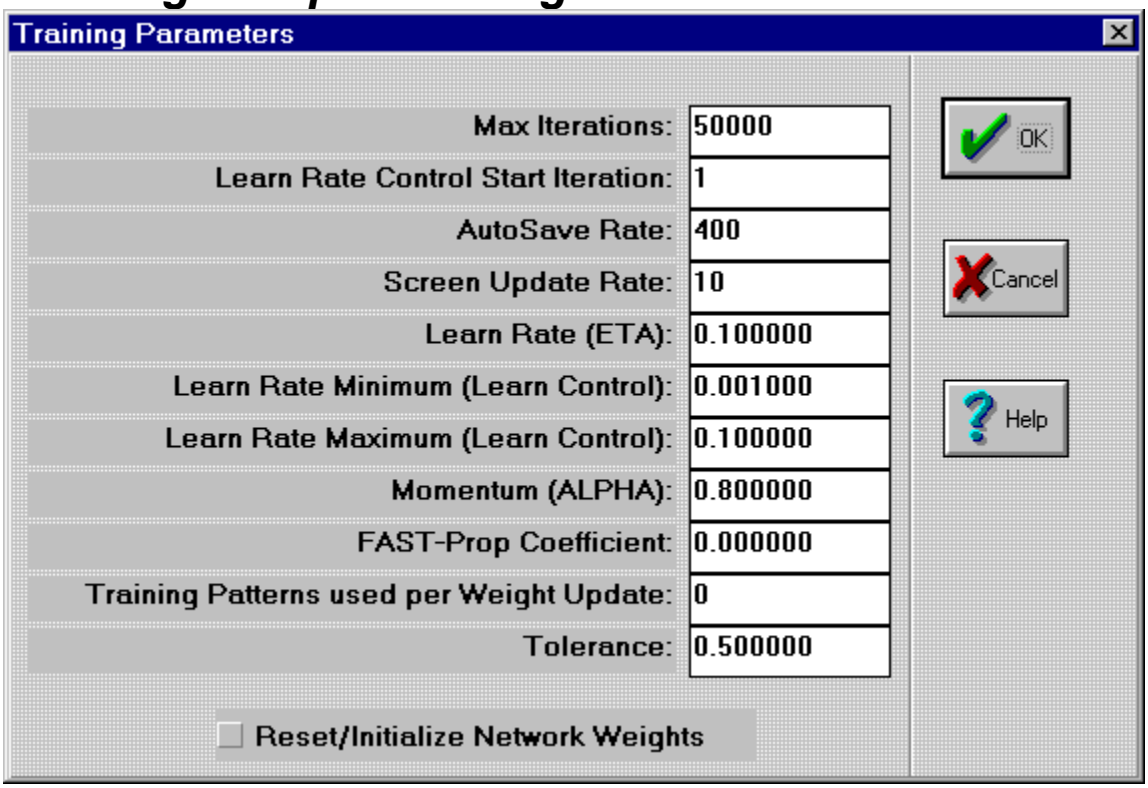
The connections in the network are shown in the left list box. The connections removed from the network are shown in the right list box. The connection notation L1, L2,... stands for Layer 1, Layer 2, etc. The notation N1, N2,... stands for Node 1, Node 2, etc. All connections between any two layers are represented by this notation and displayed in the lists. Simply select the desired connections and use the *"**remove**"* and *"**add**"* buttons to transfer the selected connections between lists. Select multiple connections by using the Shift and/or Ctrl keys in connection with the mouse pointer. Holding down the

shift key allows you to select multiple sequential connections. Holding down the Ctrl key allows you to select multiple connections individually. After editing the connections between layers, select ***"Ok"*** to register these changes. Selecting ***"Cancel"*** will cause changes to be discarded.

View Network

The "***View Network***" button will display the current design of your network.

Training Setup - Training Parameters



Number of Iterations

Enter the maximum number of iterations to perform in this training session. Training can be manually terminated before the specified number of iterations has been reached. *TIP: There is no way to predetermine the number of iterations that will be required to converge a network -- it could take a few hundred or it could take several thousand. Normally, it is best to set the number of iterations to a large value and manually terminate training at the appropriate time.*

Learn Rate Control Start Iteration

Enter the iteration number to begin Learn Rate Control (LRC). Qnet has a special algorithm to control the learning rate. This algorithm will seek the optimal learning rate range during the training run. LRC guards against divergence during training, while attempting to drive eta as high as possible. During the initial training iterations (50 to 100) for a new network, LRC should normally be turned off. The node weights of new networks are adjusted rapidly during initial iterations and the training error can oscillate wildly. Using LRC during this period is perfectly safe, however, the LRC algorithm may drive eta unnecessarily low in an attempt to eliminate these normal oscillations. (See Starting New Networks).

Learn Rate Control is less useful when the full set of training patterns are not processed per weight update cycle (see below). It is recommended that LRC be disabled (set value greater than # of iterations) when this option is activated.

AutoSave Rate

Set the rate at which Qnet's AutoSave will store the network during training. This value sets the number of iterations between stores. A rate of 100 will store the network every 100th iteration. The network is stored in a temporary file during training. If necessary, these network snap shots can be saved to permanent network definition files. This may be necessary to eliminate overtraining conditions or network divergence problems. It is recommended that the rate be set to a value that will yield network saves once every 10 to 15 minutes. The default rate is 500 iterations per save. Adjust this value during training if necessary. Stored network snap shots can be saved as permanent network definition files using the "File/Save

AutoSave File..." option in the training menu. AutoSave can be disabled by setting the rate to 0.

Screen Update Rate

This value sets the iteration interval at which screen updates occur during the training process. Updating the display every iteration with network training and convergence information provides the best visual monitoring of the training process. Unfortunately, this can negatively impact training times due to the relatively slow process of writing the information to the screen. The optimal update rate to use for monitoring the training activity depends on network size, the number of training patterns and the speed of the computer. If screen updates occur more than once every few seconds, execution speeds are being retarded. When training speeds are not a concern, set this value to 1 to provide the best interactive monitoring. The test set error (if it exists) is only computed at the screen update interval. To monitor the test data error at a reasonable interval, the report rate should be set to 10 or less.

Learning Rate (ETA)

The learn rate, eta, controls the rate at which Qnet's training algorithms attempt to learn. It determines how fast the node weights are adjusted during training. Eta's valid range is between 0.0 and 1.0. While higher eta's result in faster learning, they can also lead to training instabilities and divergence. When initiating training on a new network, the user must provide a starting eta value. It is better to start conservatively by using a low number. Using a value in the 0.001 to 0.1 range will normally start the training process safely. If the initial guess is too high and the network diverges, reset the network by selecting the "***Options/Randomize Weights***" from the training window menu. Select "***Options/Learn Rate***" from the same menu and try a lower value. Qnet's Learn Rate Control (LRC) will help keep eta in its optimal range during training when active.

Networks employing gaussian transfer functions often require lower learning rates. Since gaussian networks often train faster, a lower learning rate promotes stability without affecting overall training time.

Learn Rate Minimum

Set the minimum learning rate. Qnet's Learn Rate Control will not lower eta below this limit. The default value of 0.001 generally works well. This value may be adjusted during the training process, if required.

Learn Rate Maximum

Set the maximum learning rate. Qnet's Learn Rate Control will not raise eta above this limit. Setting this value in conjunction with LRC will help to avoid instabilities and can result in a significant improvement in convergence times. The maximum value for eta should be adjusted lower whenever training instabilities develop (while LRC is enabled). The upper stable limit of eta can be determined during training by using NetGraph to plot the eta history. A default value of 1 is set for new networks.

Momentum Factor (Alpha)

Alpha is the learning rate momentum factor used by Qnet's training algorithms. This factor promotes fast, stable learning. The valid range for alpha is 0. to 1. Most networks will learn and converge best by setting this value between 0.8 to 0.9 and leaving it there. While this is a good guideline, a different value may work better for some models. A value of 0 causes backpropagation learning with no momentum.

FAST-Prop Coefficient

The FAST-Prop coefficient controls the algorithm used by Qnet for training. If this coefficient is set to 0 (the default), Qnet will employ its backpropagation algorithm to train the network. If the coefficient is set to a value above 0.0 (to a maximum of 3.0), the FAST-Prop algorithm is used. The closer the coefficient is set to 0.0, the closer FAST-Prop approximates standard backpropagation. While the FAST-Prop training method can often accelerate the learning process, a drawback with this method is that there is a risk that this algorithm will not converge to a minimum error, especially when higher coefficient values are used. Also, this method tends to be less successful for multiple hidden layer networks and networks that employ Gaussian transfer functions. It is recommended that the training algorithm be switched to the standard backpropagation method (i.e. set the FAST-Prop coefficient to 0) at some point during the training process to improve convergence characteristics. Likewise, the FAST-Prop algorithm is NOT recommended for long periods of unattended training. Whenever FAST-Prop is being used, the training and test set RMS

errors should be monitored closely for signs that the network is no longer converging. If either of these error values begin to increase, it is recommended that the FAST-Prop coefficient be set to 0.

Patterns Per Weight Update Cycle

The number of training patterns to process per weight update cycle can have a large effect on the overall training process and convergence behavior. Qnet's default is to process all training patterns prior to updating network weights (set to 0). This allows a global error vector to be developed prior to adjusting weights. This practice generally leads to the most orderly decent of both training and test set errors at the price of slightly slower training performance. ***This default method is strongly recommended for training sets where non-precise and somewhat noisy relationships exist between the inputs and outputs***.

When weights are updated after a partial set of patterns have been processed, several distinct differences may be noted during the training process. Learn rates can play a larger factor on the observed level of generalized vs. memorization learning. Often, lower learning rates will offer more generalized learning. It is also advised that Learn Rate Control (LRC) be turned off and a low learning rate should be set that produces both adequate convergence speed and good generalized learning. The advantage of this method of network training is that weight updates are performed more frequently and network convergence times can be improved.

The number of patterns processed per weight update cycle is set in the Training Setup/Training Parameters. A value of 0 (the default) processes all training patterns per update cycle. A value of 1 processes one pattern for each update cycle, etc. This value may also be altered during training. *IMPORTANT: Since the learn rate may depend on this parameter, the learn rate should be sufficiently low to prevent divergence if this value is changed. It also advisable to save the network prior to changing this parameter.*

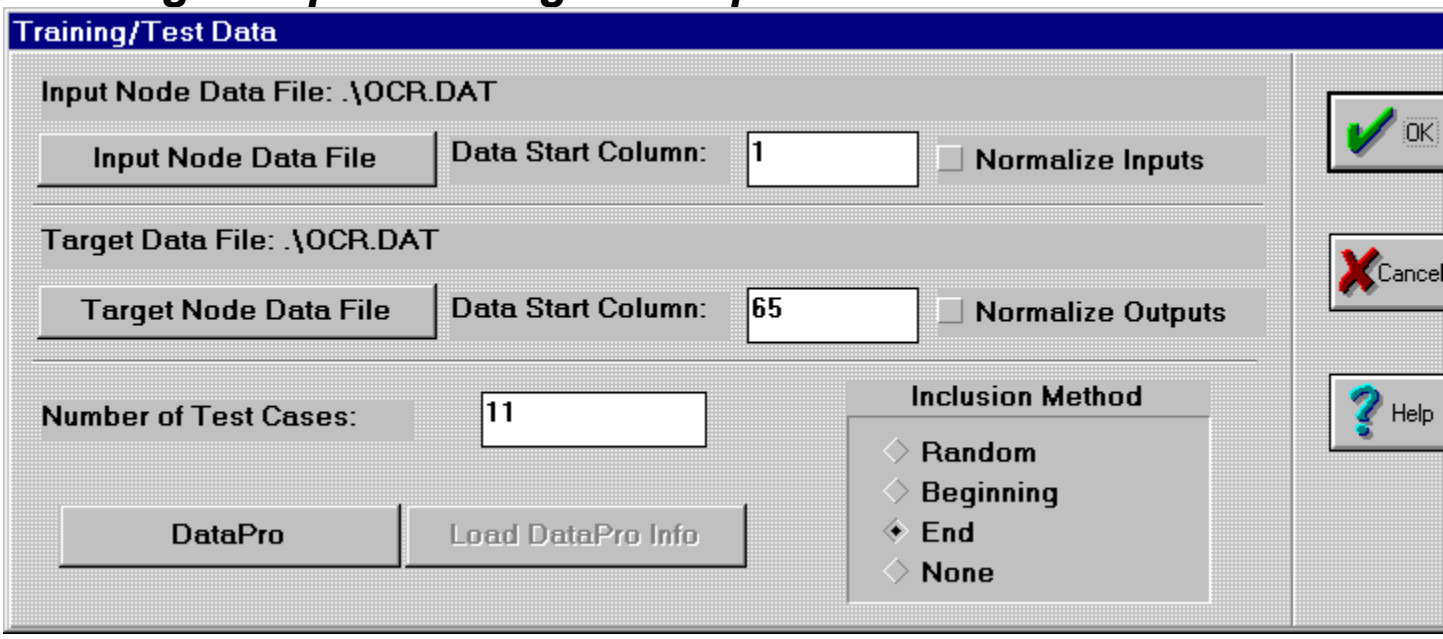
Tolerance

Selects the tolerance used to monitor training accuracy. During training this tolerance is used to determine whether the network prediction agrees with its target. The percentage of training and test cases within tolerance are displayed during training.

Initialize/Reset Weights

Node weight values represent the learned information stored in a neural network. New network weight values are initialized randomly prior to training. The "***Initialize/Reset Weights***" option is automatically selected for new networks. It can be manually selected to reset a previously trained network to an initial untrained state.

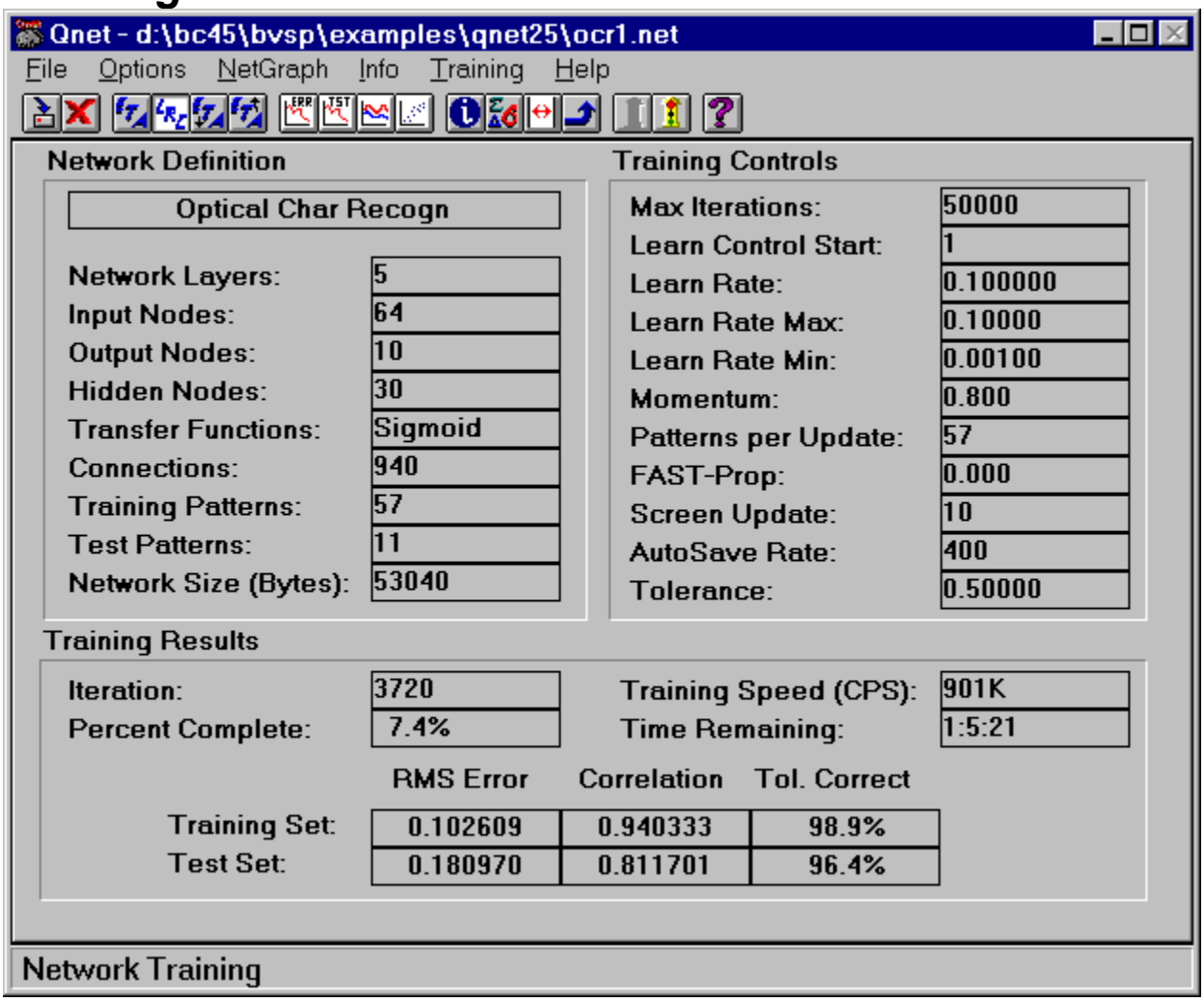
Training Setup - Training Data Specification



The training/test data dialog allows you to specify the training data you intend to use for the neural network model. The proper formats and setup procedures can be found in the training data section of this manual. The data may preexist in ASCII data files or you may use the DataPro feature to generate proper Qnet training data files. Options include:

Input Node Data
Target/Output Node Data
Data Start Column
Number of Test Cases
Test Set Inclusion Method
DataPro

Training Window



The training window is used to view, analyze and interact with the current training run. Use the ***File*** option to save training information or exit the current run. Interact with the network training parameters by using the ***Options*** menu. Check the training progress with Qnet's ***NetGraph*** and ***Info*** tools. Selecting any of these options will temporarily suspend network training as indicated on the status bar. Select ***Training*** Start/Stop to restart network training.

The information displayed in the training window provides details on the network model, the current training parameters and the training results. The ***Network Definition Group*** displays the network's name, the number of network layers, the number of input nodes, the number of output nodes, the total number of hidden nodes, the number of network connections, the number of training and test patterns, and the network size in bytes. The ***Training Controls Group*** displays the maximum number of iterations for the run, the LRC start iteration, the FAST-Prop coefficient, the learn rate settings, the momentum factor and the screen update and AutoSave rates. The ***Training Results Group*** contains the current iteration, the training and test set RMS errors, training and test set correlation coefficients and the training and test set tolerance percentages. The connections per second benchmark indicates computational speed, the percent complete, and time remaining (based on network training completing the specified number of iterations) are also included. The menu items and user options are organized as follows:

File

Save Network
Save Network As...
Save Outputs/Targets...
Save Error History...
Save AutoSave Network...
Restart Qnet
Exit

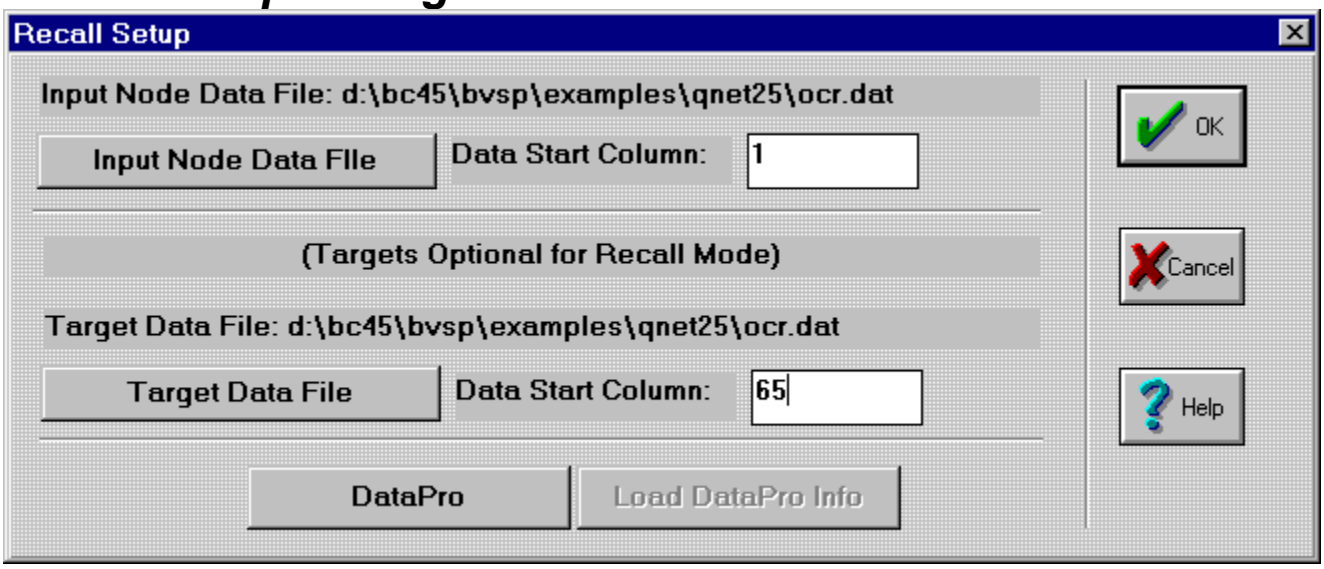
Options
Learn Rate Control
Learn Rate...
Learn Rate Min...
Learn Rate Max...
Momentum...
FAST-Prop Coef...
Patterns Per Weight Update...
Iterations...
AutoSave Rate...
Screen Update Rate...
Tolerance...
Initialize/Reset Weights...

NetGraph
RMS Error History vs Iteration...
Correlation History vs Iteration...
Tolerance History vs Iteration...
Test RMS Error History vs Iteration...
Test Correlation History vs Iteration...
Test Tolerance History vs Iteration...
Learn Rate vs Iteration...
Targets vs Net Outputs...
Targets/Outputs vs Pattern Sequence...
Output Error vs Pattern Sequence...
Input Nodes vs Pattern Sequence...
Input Node Interrogator...
Color Contours...
Node Analyzer...

Info
Network Information...
Network Outputs/Targets...
Weights and Deltas...
Statistics...
Tolerance Checking...
Threshold Checking...
Divergence Checking...

Training
Start
Stop

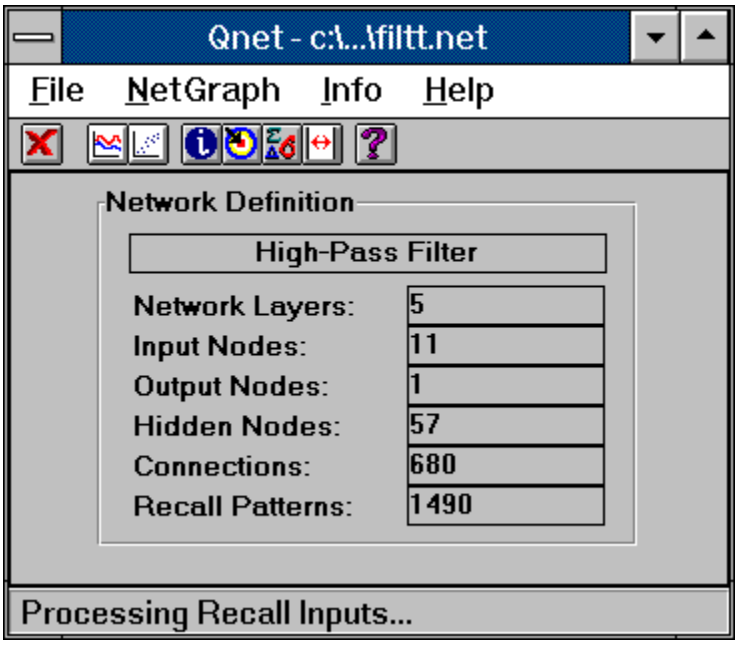
Recall Setup Dialog Window



Use the recall setup window to specify the files containing the inputs and targets (optional) to be processed through a trained network. The following inputs are used for recall setup:

Input Node Data File
Output Node Data File
Data Start Column
DataPro

Recall Window



Qnet's recall window is used to analyze a network's response to a new set of inputs. New predictions can be analyzed and/or saved to a file with the recall windows menu options.

File
Save Outputs/Targets...
Restart Qnet
Exit

NetGraph
Targets vs Outputs...
Targets/Outputs vs Pattern Sequence...
Output Error vs Pattern Sequence...
Input Nodes vs Pattern Sequence...

Info
Network Information...
Network Outputs/Targets...
Statistics...
Tolerance Checking...
Threshold Checking...

Backpropagation Technical Overview

This overview is intended to provide both general information on backpropagation theory and some specific details of Qnet's modeling techniques. While an understanding of specific theoretical details is not required for Qnet, it can provide the initiated user with a more complete overview of Qnet's internal operation.

Qnet backpropagation neural networks are multi-layered and feedforward (connections must connect to the next layer) in design. Networks can be fully connected or connections can be removed individually. Removed connections are modeled in Qnet by explicitly setting the connection's receiving weight to 0. This removes the effect of that individual connection on the network's response.

New networks have randomly initialized weight values. Each time an initialization is performed a network state will be created that is completely unique. This leads to the possibility that identical training runs with newly initialized networks may exhibit different learning characteristics. However, the converged states of two such training runs will be nearly identical for the vast majority of cases.

Backpropagation training is accomplished using the following logic sequence (NOTE: Vectors presented in Italics):

1. Input patterns are stored in an array of input vectors, $\boldsymbol{X_{(P,1)} = (Xp,1, Xp,2, Xp,3, ..., Xp,N)}$, where P is the pattern sequence number and N is the vector length (equal to the number of input nodes). These vectors are unaltered by the input layer and are output to the nodes in the first hidden layer. (NOTE: The vector elements must be pre-normalized between 0 and 1 either by Qnet or the user)

2. Each node of a given hidden or output layer receives an identical input vector, $\boldsymbol{X}$, from the preceding layer. Each node processes the vector internally through the equation:

$$\mathbf{Y_{(P,L,J)} = X_{(P,L-1)}\ \ W_{(J,L)} + B_{(J,L)}}$$

where $\mathbf{Y_{(P,L,J)}}$ is the processed result for node J in layer L (i.e. the input layer is layer 1, the first hidden layer is layer 2, etc.) The dot product is taken between the node's input vector, $\boldsymbol{X_{(P,L-1)}}$, and the node's internal weight vector, $\boldsymbol{W_{(J,L)}}$, and summed with the bias value, $\mathbf{B_{(J,L)}}$.

3. The resulting value, $\mathbf{Y_{(P,L,J)}}$, for node J is then processed through a transfer function to determine the signal strength for the node's output connection. The transfer function used by Qnet is the sigmoid function, $\mathbf{f(Y) = 1/(1+exp(-Y))}$, the gaussian function, $\mathbf{f(Y) = exp(-Y*Y)}$; the hyperbolic tangent, $\mathbf{f(Y) = (tanh(Y) + 1)/2}$; or the hyperbolic secant function, $\mathbf{f(Y) = sech(Y)}$. This function serves to normalize the output of a node between 0 and 1 and is continuous in form (the first derivative must exist for backpropagation training).

4. Each node's output value is combined in the current hidden or output layer to form the layer's output vector:

$$\boldsymbol{X_{(P,L)} = (f(Y_{(P,L,1)}, f(Y_{(P,L,2)}, ..., f(Y_{(P,L,K)}))}$$

where K is the total number of nodes in layer L. This output vector becomes the input vector to the next layer.

5. Processing proceeds to the output layer where the final output vector, $\boldsymbol{X_{(P,O)}}$ is obtained. (In recall mode processing ends at this point.)

6. The final output vector is combined with the training target vector, $\boldsymbol{T_{(P)}}$, to obtain the output layer's error vector, $\boldsymbol{E_{(P)}}$. The equation governing the computation of the error vector is:

$$\boldsymbol{E_{(P,O)} = (T_{(P)} - X_{(P,O)})\ X'_{(P,O)}}$$

where:

$$X'_{(P,O)} = (f'(Y_{(P,O,1)}, f'(Y_{(P,O,2)}, \ldots, f'(Y_{(P,O,K)}))$$ (note: **f'** is the first derivative of the transfer function **f**).

The error for node J in hidden layer L is computed by the equation:

$$E_{(P,L,J)} = X'_{(P,L,J)} * SUM_K(E_{(P,L+1,K)} * W_{(K,L+1,J)}).$$

where K represents the Kth node in layer L+1. Through this method, the error vector, ***E(P,L)***, is obtained for each hidden layer. Note that this equation causes the errors to be backpropagated through the network (thus the name for the paradigm).

7. Next the weight vectors for each node must be updated. The new weights for node J in layer L (output and hidden) are computed by:

$$W_{(J,L)\,T+1} = W_{(J,L)\,T} + (eta)\, E_{(P,L,J)}\, X_{(P,L-1)} + (alpha)\,(\, W_{(J,L)\,T} - W_{(J,L)\,T-1}).$$

where (eta) is the learning rate, (alpha) is the momentum factor and T is the iteration cycle. Note that the weight change computed from the previous weight update cycle is multiplied by the momentum factor. The momentum term helps to keep the training process stable by damping weight change oscillations.

8. All input vectors (patterns) are processed through the network to adjust the weights for a given iteration.

9. The RMS error between the network response and the training targets is computed by Qnet after each iteration. Its equation is given by:

$$RMS\ Error = SQRT(\, SUM_{P,K}((T_{(P,K)} - X_{(P,O,K)})\text{^}2)/(P_T * K_T))$$

where P is the Pth input pattern and K is the Kth output node. P_T is the total number of patterns and K_T is the total number of output nodes. The RMS error is also equivalent to the standard deviation of the error in the network's response.

10. If Learn Rate Control is active for the run, a new learning rate, (eta), is computed by Qnet based on the change in the RMS Error value.

11. The entire process cycles again with next training iteration.

The FAST-Prop method used in Qnet differs by the weight update algorithm. The modified to the form of this equation becomes:

$$W_{(J,L)\,T+1} = W_{(J,L)\,T} + (eta)\, E_{(P,L,J)}\, (X_{(P,L-1)} + (fp)\, E_{(P,L-1)}) + (alpha)\,(\, W_{(J,L)\,T} - W_{(J,L)\,T-1})$$

where (fp) is the FAST-Prop coefficient.

Qnet Limits and Specifications

Qnet is designed to handle extremely large and complex network designs with no artificially imposed software restrictions. However, practical limits will exist for both network design and training set sizes. These numbers can vary significantly and will depend largely on your hardware's processing speed and available memory. Qnet's single limit on the size of your neural network design is that you may use a maximum of 10 layers. This maximum easily satisfies all conceivable modeling requirements.

Tech Support

Tech support is available to registered users at (847) 446-1655 or via the internet at VestaServ@aol.com.

Steps for Creating a Neural Network Model

If you're new to neural networks, creating your own neural model may be somewhat of a mystery. We strongly recommend that you first familiarize yourself with Qnet's menus and training options by investigating a few of Qnet's sample networks prior to setting up and training your own models. The following steps provide a detailed overview of the process required for generating successful neural network models:

Conceptualize the model: Formulate the idea for your model. Whatever type of model you intend to create, you must first conceptualize the inputs and outputs to be used. First decide what it is you want to predict or determine with the model. The information that is to obtained from the model may be one simple output or it may be many pieces of information (multiple outputs). Next, decide what factors influence the output(s). What information should the network use to learn the problem and predict the answer? The inputs could be many separate pieces of information or a single set of information that may be supplied through many input nodes. For example, a visual recognition problem requiring pictures as input, will require that each image be broken down and compressed in some manner to adequately process input information. Other models may start with a relatively few number of raw inputs, but can be improved and advanced by preprocessing and reformulating the inputs to create a more sophisticated input set. Properly conceptualizing how the inputs and outputs are to be formulated is the first major step in building a successful model.

Gather the data: After conceptualizing the model, the training data must be gathered. This is often the most tedious task in model development. If you have access to all the needed information through existing databases, you're in luck. Otherwise information must be accessed through other information resources. Common sources of data include: On-line information services, CD ROM's, scanned data from references, experimental data captured through data acquisition systems, computer generated data, purchased data and manual data entry.

Process and format the data: Once the training data is available, it is often easiest to use your system's spreadsheet application to properly setup, preprocess and format the data. Most spreadsheets will allow you to import data from multiple sources and in a variety of formats. Once the data has been collected, you should determine what inputs need to be preprocessed. Are some inputs unbounded? Are they using implied rankings? Which inputs should be represented in a binary format? Which inputs should be represented in a continuous value format? Determining the proper answers to these questions is one of the most critical steps in developing a good model. For most data reformulating tasks, a good spreadsheet will have all the tools necessary to effectively handle the job. After the data is in the desired form, finalize the file format by grouping the inputs and the targets in adjacent columns. If data labels are desired, they can be added at this time also. You may either save (or export) the properly formatted data to an Qnet compatible ASCII file or use Qnet's DataPro to directly transfer data from a Window's based spreadsheet into Qnet.

Create the neural network model: With the training data now created (or available for transfer), it's time to setup the model with Qnet. Start Qnet and select "RUN" to activate the Training Setup Dialog and define the model. It's usually be best to start with a simple network/hidden layer construction so that you can quickly validate the model's ability to train and learn relationships. Specify the network construction, the training data files, and your initial "best guess" at what the training parameters should be (remember - the default training parameters work for the vast majority of problems and all parameters are adjustable during training). To check and validate your training data files not created by DataPro, use the utility to load and scan your data for correct formatting. New networks should be saved prior to starting training by selecting Save Network Setup. (NOTE: No setup information is saved until this save option is selected or the network is saved during training.)

Train the network: Use Qnet's analysis tools to monitor network training. Tools such as NetGraph and the various training Information options provide insight into how the network is training, when convergence has been achieved and information on the quality of the model being generated. It is also

likely that you will need to tune the training parameters so that the specified values match the training characteristics for your model. You may optionally terminate training and restart the training at any point.

Analyze the trained network: When training is complete (or near complete) use Qnet's input interrogator to look at effectiveness of each input in predicting the output(s). The node analyzer tool can be used to determine if specifying more or less nodes might improve the model. Examine how the network performs with your test data. How your model performs with test data not used to directly train the network will be the best indication of how the model will perform under real life conditions. These analyses will provide you with information for both how the model will perform and whether you might be able to improve the model by making design changes. Print and save the final network configuration and statistical comparisons if you plan on trying different network configurations. This provides a way of quickly comparing different designs.

Try additional designs: After the network training and analysis has completed for the first modeling attempt and you've determined that additional model designs should be explored, you should revisit either the **Gather the data, Process and format the data** or **Create the neural model** steps. The step you return to depends on whether you wish to augment or improve the input data or simply wish to explore additional or more complex network designs. To investigate additional designs, simply open the first Qnet network definition file, make the necessary changes in the training setup dialog windows, save the network under a new network name and start training. This will preserve the original network and allow you to investigate new designs without overwriting previous results.

Accessing trained neural networks: Trained networks can be accessed in two ways. Qnet's recall mode is one quick and easy method for accessing trained models. If you intend to access the model frequently, consider automating access with QnetTool. Alternatively, programmers can incorporate trained networks directly into applications using Qnet's C source code.

With a little experience you'll find that creating neural network models can be relatively easy. The pace you proceed at will be dictated by your experience with neural networks and your experience with handling and processing data.

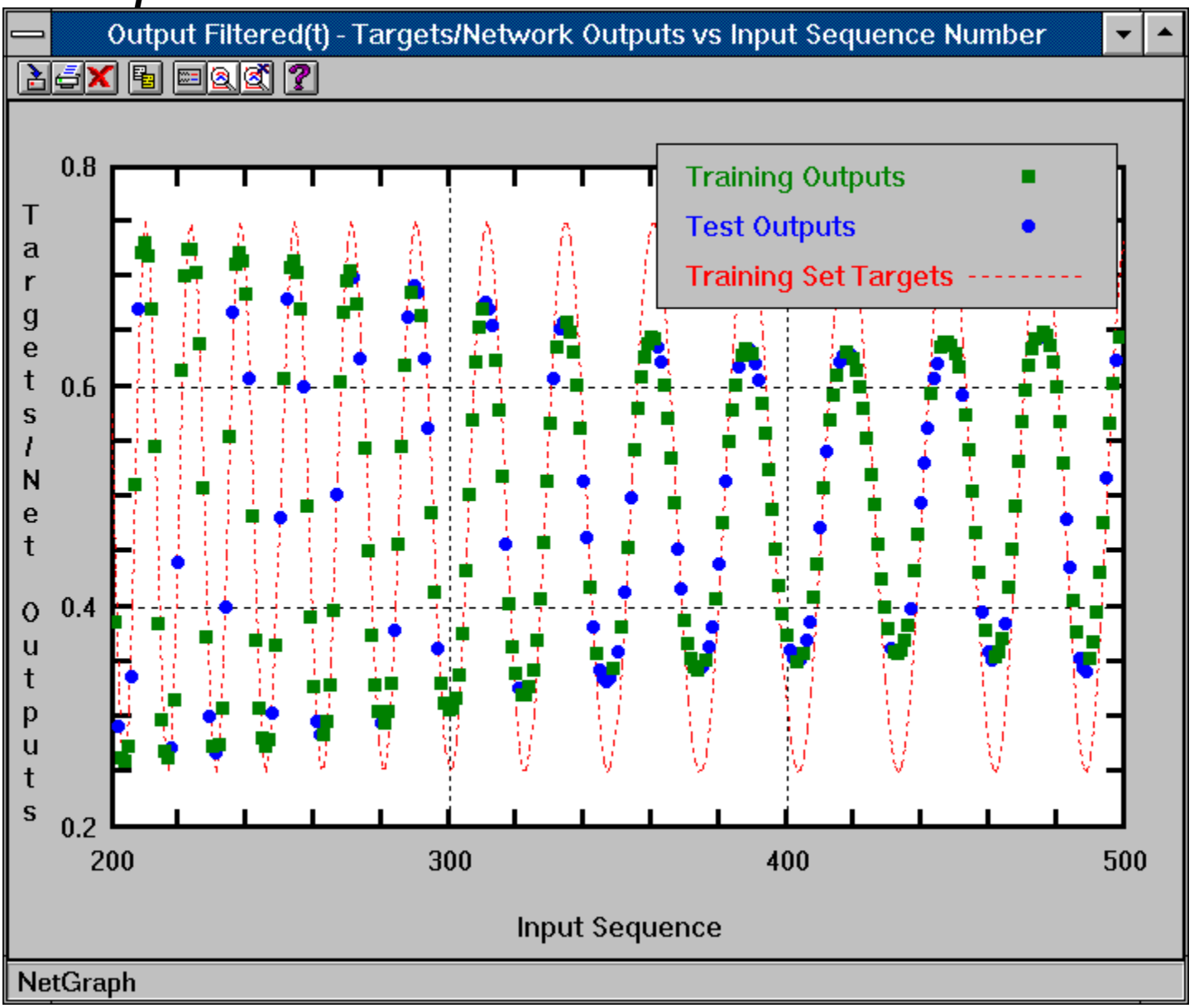
QnetTool

Creating advanced neural network models is made easy with Qnet. However, to productively utilize your neural network models, you must have the ability to incorporate them directly into your everyday work. Qnet gives you the standard recall mode access and includes C source for programmers to incorporate neural networks into new applications. Unfortunately, for most of us, this still doesn't provide us with the means to fully integrate our neural network models with our everyday work. QnetTool gives us this ability. Seamless integration can be accomplished even by novice users. No complicated data preparation procedures or advanced programming skills are required.

With QnetTool you can now integrate your neural network solutions with your favorite Windows spreadsheet or database application. Multiple levels of integration are available depending on your automation needs. Manual integration of QnetTool with your Windows applications involves simple copy/paste procedures between the data application and QnetTool. Inputs can be copied to QnetTool and network solutions can be copied back to your data application with simple menu/button selections. Automatic integration involves the creation of simple macros in your data application that give you complete hands-off retrieval of Qnet solutions. Imagine, click a button in your spreadsheet and have your neural model answers loaded and usable immediately. Examples are included for both Excel and Quattro Pro that demonstrate how short, 10 line macros can do just that. QnetTool provides quick and total integration of your neural network solutions with your data analysis application.

If you don't currently have QnetTool, call Vesta Services, Inc. for more information (708-446-1655).

NetGraph



Qnet's graphing tool, NetGraph, can be used to visually monitor and check the training history and the quality of the model being developed. Use NetGraph's AutoZoom feature to instantly zoom into any portion of the plotted data. Any graph can be saved in a Windows bitmap (.BMP) file or copied to the clipboard for use in drawing programs, word processors, etc. These functions can be accessed by pressing the toolbar buttons:

SAVE
Save a plot in a bitmap/BMP file format (requires video display mode to be in 256 color mode or less).

PRINT
Send the plot to the printer.

EXIT
Exit the current plot.

COPY
Copy the current plot to the clipboard.

LEGEND ON/OFF
If a legend is present with the plot. You may wish to remove the legend temporarily to better view the plot. Selecting this button will toggle the legend's state between visible and invisible.

ZOOM
Invoke AutoZoom mode. AutoZoom works by pressing the Zoom button and then dragging the mouse with the left mouse button held down to mark the area to zoom into. The zoom box should be marked from the upper left to the lower right corner. Once the zoom box is set, the plot is redrawn to view the area selected.

RESET
Reset a zoomed plot to the originally scale by pressing the Reset button.

To reposition the legend on the plot, simply double-click the left mouse button. This will cause the legend to reposition at the current mouse position.

AutoZoom works by selecting the Zoom button and then dragging the mouse with the left mouse button held down to mark the area to zoom in on. The zoom box should be marked from the upper left to the lower right corner. Once the zoom box is set, the plot is redrawn to view the area selected. The reset button will redraw the zoomed plot to the original scale.

Info Browser

Qnet's Information Browser can display general design and training information, network node weights, tolerance and threshold check results, training outputs and targets and statistical comparisons. The following options are available in Qnet's Information Browser:

Save

Save the information to an ASCII (text) file. Specify the path/filename in the displayed file dialog.

Print

Send the information to the printer. Specify printer and print options from the displayed dialog.

Exit

Exit the Information Browser.

Overtraining

Overtraining occurs when the test set error increases while the training set error continues to descend. This indicates that memorization is the predominant learning mode. When a test set error has reached a global minimum and increases indefinitely thereafter, overtraining has occurred. Training a network after the test set error global minimum has been reached can actually hurt the predictive capabilities of the model being developed.

network definition file

The network definition file is Qnet's file that contains information on the network design, training options, input/output normalization values and the training weights. The network definition file is opened at Qnet's startup dialog and may be saved from the training window and/or the training setup dialog. The default file extension is **".NET"**.

FAST-Prop coefficient

Set in the Training Setup/Training Parameters dialog or during training, this factor alters the weight update formula used for backpropagation training. See Technical Overview and Backprop vs FAST-Prop for more information.

continuous valued
Data represented by integer (0,1,2,...) or floating point (1.2,2.33,3.52) numbers.

binary
A switch representing on/off, yes/no, etc. Usually presented to a network as 1's and 0's in the training data.

DataPro

DataPro allows you to edit or create Qnet training data files. DataPro's spreadsheet editor lets you alter existing training files or create new files from scratch. New Qnet training data files may be created by pasting data from existing spreadsheets into DataPro or by direct entry. For more information see the DataPro's Help file.

After a file is created with DataPro, use the "***Load DataPro Info***" button to fill in the Dialog with the appropriate responses.

Learn Rate Control

Qnet's automated learn rate control can be used to alter the training learn rate factor during training in an attempt to optimize network convergence. See Learn Rates and Learn Rate Control for more information

Learn Rate Coefficient

The learn rate coefficient (ETA) is a factor used in Qnet's backpropagation training algorithm and controls how large the weight update corrections are during each update cycle. Smaller values make for more accurate and stable updates, however training times can become very long. Larger values can dramatically improve convergence time, but when the learning rate become too large, weight updates become less accurate and can lead to training instabilities or complete training divergence. The maximum stable learn rate depends on many factors. These include: the training set size and overall data trends, the values of other training parameters and the neural network construction parameters. Qnet includes a Learn Rate Control feature that will attempt to keep the learn rate near the maximum stable range. The other alternative is to set the learn rate to "safe" low value that will allow the network to train using a constant factor. The valid range for the learn rate coefficient is between 0. and 1.

The learn rate is set in the Training Setup/Training Parameters dialog.

momentum factor

The momentum factor, alpha, is used by Qnet's backpropagation algorithm to damp oscillations in the weight update process. The momentum factor attempts to drive the weight update in the correct (i.e. the same) direction each weight update cycle in an effort to improve convergence charactistics and eliminate training instabilities. The valid range for the momentum factor is between 0. and 1. The momenum factor can be set in the Training Setup/Training Parameters dialog.

Patterns Processed per Weight Update Cycle

The number of patterns to process per weight update cycle can have a large effect on the overall training process and convergence behavior. Qnet's default is to process all training patterns prior to updating network weights. This allows a global error vector to be developed prior to adjusting weights. This practice generally leads to the most orderly decent of both training and test set errors at the price of slightly slower training performance. This method, however, is strongly recommended for training sets where non-precise and somewhat noisy relationships exist between the inputs and outputs.

When weights are updated after a partial set of patterns have been processed, several distinct differences may be noted during the training process. Learn rates can play a larger factor on the observed level of generalized vs. memorization learning. Often, lower learning rates will offer more generalized learning. It is also advised that Learn Rate Control (LRC) be turned off and a low learning rate should be set that produces both adequate convergence speed and good generalized learning. The advantage of this method of network training is that weight updates are performed more frequently and network convergence times can be improved.

The number of patterns processed per weight update cycle is set in the Training Setup/Training Parameters. A value of 0 (the default) processes all training patterns per update cycle. A value of 1 processes one pattern for each update cycle, etc. This value may also be altered during training. IMPORTANT: Since the learn rate may depend on this parameter, the learn rate should be sufficiently low to prevent divergence if this value is changed. It also advisable to save the network prior to changing this parameter.

Input Node Data File

Select the Input Node Data file button to bring up a file selection dialog to specify the file containing the training/recall data for the input nodes. The file format should be an ASCII (text) file in columnar format. This is the type of format that most spreadsheets and database applications produce if row/column cell values are written out in text or ASCII mode. Data columns in the file are mapped to the input nodes. Each row represents a separate training/recall case (i.e., one pattern). Spaces, commas or tabs may be used to separate the data columns. Blank rows and rows starting with a "#" are ignored. (NOTE: Inputs and targets may be contained in the same or separate files.)

After selecting the file, the DataPro button allows the input data to be examined or edited.

Data Start Column

Training and recall data must be in a columnar format. Each input node will use one column of data. Data columns may be delimited by commas, spaces or tabs. The value specified in this field is the data column where input node 1 starts. For example, if the network has 50 input nodes and the file contains the input node information in data columns 2 through 51, enter 2 in this field. The input node data columns must be contiguous (i.e., the data cannot be in non-contiguous columns 2, 5, 9, ..., etc.).

Output Node Data File

Select the Output Node Data file button to bring up a file selection dialog to specify the file containing the target data for the output nodes. The file format should be an ASCII (text) file in columnar format. This is the type of format that most spreadsheets and database applications produce if row/column cell values are written out in text or ASCII mode. Data columns in the file are mapped to the output nodes. Each row represents a separate training/recall case (i.e., one pattern). Spaces, commas or tabs may be used to separate the data columns. Blank rows and rows starting with a "#" are ignored. (NOTE: Inputs and targets may be in the same or separate files.)

After selecting the file, the DataPro button allows the targets to be examined or edited.

Save Outputs/Targets...

Save the network outputs and targets (if available) in ASCII (text) row/column format with tabs used for data column delimiters. The format is:

<pattern 1> <node 1 target> <node 1 net output> <node 2 target> <node 2 net output> ...
<pattern 2> <node 1 target> <node 1 net output> <node 2 target> <node 2 net output> ...
etc...

If, in recall mode, no targets were presented to the network, only network predictions will be available.

Restart Qnet

Return to Qnet's startup window. This option is available from either the training or recall windows.

Exit
Exit Qnet.

Targets vs Outputs

Plot the network outputs versus the target values in a scatter plot format. This graph format quickly compares how well all outputs and targets agree for all training patterns. The values plotted are the normalized (between 0 and 1) targets and outputs. The closer all plotted points fall on the line "Y=X", the better the agreement between network output and training targets.

Targets/Outputs vs Pattern Sequence...

Plot the target values and network output data versus the input pattern sequence number. Both training and test data are plotted on the graph. Separate graphs are generated for each output node. NetGraph will cycle through all output nodes or allow you to select a specific output node.

Output Error vs Pattern Sequence

Plot the difference between the target and network output data versus the input pattern sequence number. Separate graphs are generated for each output node. NetGraph will cycle through all output nodes or allow you to select a specific output node to plot.

Input Nodes vs Pattern Sequence

Plot the network inputs versus the input pattern sequence number. A separate graph is generated for each input node. NetGraph will cycle through all input nodes or allow you to select a specific node to plot.

Network Information

View general information about network construction and the current training status. This information can be saved or sent to the printer for a permanent record of network design and training status. This can be particularly useful when comparing different network designs for a particular problem.

Network Outputs/Targets

Select this item to view network outputs and training targets. Patterns that are used for testing (if any) are denoted with a "*" after the pattern sequence number.

Statistics

Select this item to view statistical comparisons between the training targets and network output. Statistical comparisons are made for all output nodes and for both training and test sets. The standard deviation of the error between the outputs and targets is computed, along with the bias between the two sets. Bias terms close to zero indicate that the error in network predictions are normally distributed and do not contain a systematic bias. The maximum error is shown and the correlation coefficient between the network output and targets. The closer the correlation coefficient is to a value of 1, the better the outputs predict the targets. Values near 0 (or below) indicate that little association exists between targets and outputs. These statistical measurements become more reliable as more training and test patterns are used. For very small training and test sets, these statistical measurements will be unreliable.

Tolerance Checking

This option is useful in determining the number of network predictions that fall within a selected tolerance from the training targets. This option will count the number of points that fall within the tolerance and total them in the "Correct" column. Points falling outside the tolerance are totaled in the "Wrong" column. Separate computations are made for both training and test sets.

Threshold Checking

This option is beneficial for analyzing network models with a particular type of output characteristic. Networks that are modeling an output that is essentially an up or down prediction (for example, a financial model that is predicting percent gains and losses) can use this option to gain insight into the quality of the network model. A positive, negative or combination of both threshold values are specified and only network predictions that exceed these thresholds (+ or -) are counted "Correct" or "Wrong". "Correct" and "Wrong" is determined by the direction change of the actual target. If both the prediction and target move in the same direction (i.e. have the same sign) and the prediction exceeds the threshold, a "correct" case would be counted. "Wrong" cases occur when the predicted direction is not correct. If the network prediction does not exceed the threshold, the case is ignored. By analyzing results of several threshold values, one can determine the point that the network model begins to yield reliable predictions. For models that do not have this type of output format, this option provides no useful information.

Save Network

Save the current network setup and training results in the existing network definition file (.net file). ***THIS OPTION MUST BE SELECTED TO SAVE BOTH SETUP AND TRAINING RESULTS. IF A FILE IS NOT SAVED DURING TRAINING OR SETUP, LOSS OF ALL SETUP AND/OR TRAINING ACTIVITY WILL RESULT!***

Save Network As

Save the current network setup and training results in a new network definition file (.net file). ***A NETWORK SAVE OPTION MUST BE SELECTED TO SAVE BOTH SETUP AND TRAINING RESULTS. IF A QNET NETWORK DEFINITION FILE IS NOT SAVED DURING TRAINING OR SETUP, LOSS OF ALL SETUP AND/OR TRAINING ACTIVITY WILL RESULT!***

Save Error History

The training run's error, correlation and tolerance histories are kept in a temporary file during the training run. Use this option to save the training and test histories to an ASCII (Comma Sep. Values - CSV) file for further use. The saved file will contain the training and test root-mean-square (RMS) error, correlation coefficients, and tolerance percentages for each screen update iteration.

Save AutoSave Network

Create a permanent network definition file from the AutoSave snap shots stored during training. Use this option to recover from network overtraining or training divergence conditions. You will be prompted for the name of the network definition file (you may use the same or a new filename) and you must select the iteration number from the list of AutoSave available iterations. To continue training with this newly saved Qnet network definition file, select the Restart Qnet option and open this file (if necessary).

Learn Rate Min/Max

Select these items to set the minimum and maximum learning rates to be used by Qnet's Learn Rate Control. The learn rate, eta, will be confined to these limits (valid range between 0 and 1). Setting this value in conjunction with LRC will help to avoid instabilities and can result in a significant improvement in convergence times. The maximum value for eta should be adjusted lower whenever training instabilities develop (when LRC is enabled) at frequent intervals. The upper stable limit of eta can be determined during training by using NetGraph to plot the eta history.

Iterations
Set the maximum number of iterations to use in this run. Specify a limited number of iterations or set this option to an arbitrarily high value and terminate training interactively.

AutoSave Rate

Set the rate at which Qnet's AutoSave feature will store the network during training. This value sets the number of iterations between stores. A rate of 100 will store the network every 100th iteration. The network is stored in a temporary file during training. If necessary, these network snap shots can be retrieved and saved to permanent network definition files. This is necessary to eliminate overtraining conditions or network divergence problems. It is recommended that the rate be set to a value that will yield network saves once every 10 to 15 minutes. A stored network snap shot is saved to permanent network definition files using the ***File***/Save AutoSave Network... option in the training menu. AutoSave can be disabled by setting the rate to 0.

Screen Update Rate

This value sets the iteration interval at which screen updates occur during the training process. Updating the display every iteration with network training and convergence information provides the best visual monitoring of the training process. Unfortunately, this can negatively impact training times due to the relatively slow process of writing the information to the screen. The optimal update rate to use for monitoring the training activity depends on network size, the number of training patterns and the speed of the computer. If screen updates occur more than once every few seconds, execution speeds are being retarded. When training speeds are not a concern, simply set this value to 1 to obtain the best interactive monitoring. The test set error (if it exists) is only computed at the screen update interval. To monitor the test data error at a reasonable interval, the report rate should be set to 10 or less.

Initialize/Reset Weights

Select this item to reset network weight values to a randomly initialized state. All previous training will be lost. This option is the default for new networks, but may be selected to reset networks that have diverged or trained poorly with previously set training options.

Training RMS Error History vs Iteration

Plot the run's training error history. The error is the root-mean-square (RMS) error between training set targets and network outputs. The goal of Qnet's training algorithm is to drive the error to a minimum value. Often oscillations and large changes in the error value's magnitude will make viewing changes in the current error value hard to detect. Use NetGraph's AutoZoom to better view any portion of the error history.

Test Set Error History vs Iteration

Plot the test data's error history. The error is the root-mean-square (RMS) error between network outputs and targets in the test set. Use this option to examine how well the network model predicts cases outside the training set and determine the overtraining status. If the test data's error is declining, constructive learning is occurring. TIP: Use NetGraph's AutoZoom to better view portions of the test data's error history. Oscillations in this parameter can sometimes make it difficult to view the error history with the original scale.

Learn Rate vs Iteration

Plot the Learning Rate history for this run. If LRC is on, it is often helpful to examine how the learn rate (eta) is changing. This plot can help you determine the current maximum stable learn rate . Limiting eta to the current maximum stable value will promote faster training.

Training Start/Stop Toggles

Network training may be interactivity started and stopped by selecting these options. Network training is automatically halted when you perform any analysis or training control option (except LRC toggle). Training must be manually restarted (Training/Start) after completing the desired function(s).

Divergence Checking

If a training divergence occurs during training, the diverged output node produces network predictions approaching all 0's or 1's (normalized). This check scans the network output node predictions for this behavior and indicates any suspect nodes. Any indicated output node should be plotted to verify the diverged condition.

Weights and Deltas

View the node weights and the current correction factors being used to adjust them. At the nodes where connections have been removed, the weights are set to zero.

Input Node Interrogator

After a network is near its fully trained state, it is often useful to determine what inputs are important to a network's output response. For each output node, a plot will be produced showing the relative importance of each input on that particular output. Sensitivities are determined by cycling each input for all training patterns and computing the effect on the network's output response. Please note, this method of computing sensitivity assumes that each input value is independent of all other inputs. For models where this is not true, some caution should be used when interpreting the results.

Node Analyzer

This plot helps determine how the hidden nodes are being utilized by the network. For networks that are over designed in the hidden layer structure, many nodes may contribute little or nothing to the output response. For each hidden layer in the network, a plot will be generated comparing the relative strengths of all output connections for that layer. The plot shows the nodes' percent contribution to that layer's output signals over all training patterns. If there are many nodes in a layer that are showing limited contributions, then that layer likely has too many nodes. Likewise, if all nodes show strong contributions then it is possible the adding extra nodes would help the model.

Correlation History vs Iteration

Plot the run's training set correlation history. The correlation coefficient measures the how well the network predictions trend with the targets. This plot is available during training to analyze the agreement quality. Typical models will see this value trend from near 0 towards a maximum of 1 for models that exhibit close agreement.

Tolerance History vs Iteration

Plot the training set's tolerance history. The tolerance history is the percentage of network predictions that fall within the defined tolerance. The history can indicate both the progress of training and it can help to quantify when the training has reached the required level of learning.

Tolerance Value

Selects the tolerance used to monitor training accuracy. During training this tolerance is used to determine whether the network prediction agrees with its target. The percentage of training and test cases within tolerance are displayed during training.

Number of Test Cases

A portion of the training files can be set aside and used for both overtraining analysis and for checking the quality of the neural model. To use part of the data for these purposes, give the total number of points to use and also tell Qnet how to select these points (using the selection list). It is suggested that at least 10 percent of the total number of training points be used for testing. You may select more or less at your preference, however, Qnet limits you to using, at most, 60% of the total number of available training points.

Test Set Inclusion Method

When a test set is to be incorporated into the training process, Qnet must select which patterns to set aside for testing. You may instruct Qnet to select the patterns randomly from the training file or simply use the beginning or ending specified number of patterns from the file. If the training data is in some type of systematic order, selecting the "random" method will normally offer the best results. If the training cases are not organized in some particular way that would effect the training/testing results, selecting the points from the beginning or ending of the file is also appropriate. For the "random" method, the points will be selected once. Subsequent training session using the same input data with the same number of test cases will result in the same random cases being used. If the total number of training cases changes or if you later select a different number of test set cases, a new random selection will occur. *(NOTE: For very large training sets that contain more than 32768 cases, the "random" selection method will only use the first 32768 patterns when formulating the test set cases.)*

Optical Character Recognition

Example Problem:

Network Name: Optical Char Recogn
Number of Layers: 5
Connections: FULL
Input Layer:
- **Nodes:** 64
- **Transfer Function:** Linear

Hidden Layer 1:
- **Nodes:** 10
- **Transfer Function:** Tanh

Hidden Layer 2:
- **Nodes:** 10
- **Transfer Function:** Sech

Hidden Layer 3:
- **Nodes:** 10
- **Transfer Function:** Gaussian

Output Layer:
- **Nodes:** 10
- **Transfer Function:** Sigmoid

This example is designed to illustrate the effectiveness of using Qnet to develop optical recognition applications. The optical character recognition (OCR) field is growing rapidly due to the proliferation of FAX modems and optical scanners. OCR software allows the user to turn scanned or faxed graphic images into usable text. Advanced OCR applications have recently started to employ neural networks to perform the character recognition task. Neural OCR models can greatly improve the translation accuracy over less sophisticated methods.

We will set up a small example to show how a neural network can be designed to recognize characters. The numbers 0 through 9 will make up the character set for this model. A full featured OCR program can normally recognize 100 or more characters and symbols. The characters in this example will consist of 8x8 bitmap images. This means that a total of 64 bits will be used to draw the image of each number (8 bits across by 8 bits down). Several different images (or font types) will be used for each number so that we can teach our neural network a variety of possible number types. For example, there is more than one way to draw the number “4” and we want the neural model to successfully handle the different possibilities. The training set, therefore, will consist of multiple bitmap images of the numbers 0-9. The test set for this problem will consist of number images slightly different from the numbers in the training set. This will enable us to determine how well the network has learned to generalize the differences between each number. If the network can only recognize character images that exactly match the ones in the training set, the model would not be useful in an OCR application that must process many different and imperfect character images.

Some of the bitmaps for numbers 1 through 5 are shown on the following page. The test cases used for numbers 1 through 5 are also shown. For each number, 64 inputs are generated for our neural model. Every bit that is turned on in a character’s bitmap pattern has a value of 1 and each bit that is off has a value of 0. An input array of 64 1’s and 0’s will make up each character used in the training (and test) set. As a result, the network design for this model must consist of 64 nodes in the input layer. The output layer has been designed with 10 output nodes—one node for each of the characters we wish to recognize. When a number is recognized from a set of inputs, the network will output a 1 at the appropriate output node. For this model, when the number 1 is recognized the first output node will be 1, the second output node will be 1 when the number 2 is recognized and so on (see section 8.2 for a discussion of binary mode output). The hidden structure has 3 layers containing 10 nodes each. Data normalization is not necessary for this model since all input node data and training targets use a binary representation. The network is fully connected and uses a hybrid transfer function structure.

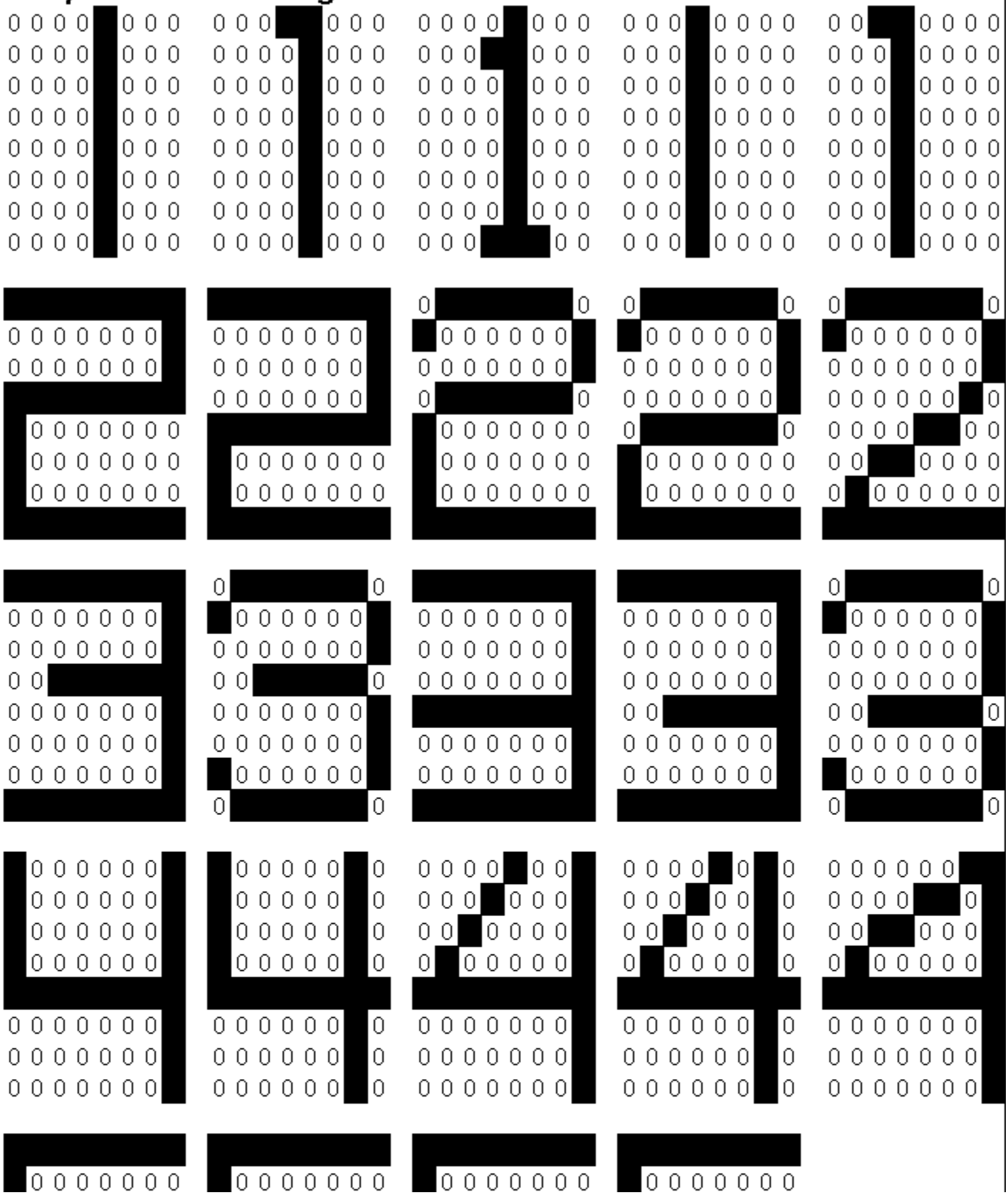
The training set is contained in the file ***OCR.DAT***. The file contains 68 total patterns (58 used for training and 10 used for the test set). Data columns 2 through 11 contain the training targets and 12 through 75 contain the input node bitmap data. The Qnet network files ***OCR.NET*** (the untrained network) and ***OCRT.NET*** (the trained network) are available to run. Use NetGraph to visually analyze the quality of

agreement between the model's output response and the training targets.

The training results indicate that the Qnet neural model easily learned to correctly classify the bitmap images of the training set. Each of the 10 test cases were also correctly recognized by the network. This indicates that the learning was generalized enough that it could classify non-learned, similar case types to a good degree of accuracy.

To build a character recognition model for a full featured OCR application, the neural net model must be significantly more sophisticated than our sample shown here. There would likely be 100 or more output nodes to properly classify most of the common characters. Advanced capabilities like font type detection can be added. The 8x8 bitmap pattern used to represent a character in this model is too small. A better representation would be 16x16 or higher. More sophisticated methods of handling input bitmaps like using 1 or 2 bytes of bitmap data per input node should be considered to reduce network size and increase efficiency. If 10 to 20 different font types were used in training, along with imperfections in these fonts (i.e., slightly rotated or non-centered), we would likely have a training set of 5000 to 10000 patterns. To adequately learn all this information, a large complex hidden structure would be necessary. While a.full OCR example is beyond the scope of our sample problems here, it is completely within the capabilities of Qnet to handle such modeling tasks.

Sample of OCR Training Cases:



Digital Filter

Example Problem:

Network Name: High-Pass Filter
Number of Layers: 5
Connections: FULL
Input Layer:
  Nodes: 11
  Transfer Function: Linear
Hidden Layer 1:
  Nodes: 20
  Transfer Function: Tanh
Hidden Layer 2:
  Nodes: 15
  Transfer Function: Tanh
Hidden Layer 3:
  Nodes: 10
  Transfer Function: Tanh
Output Layer:
  Nodes: 1
  Transfer Function: Sigmoid

This example illustrates how a backpropagation neural network can be used for noise removal from a time series signal. A simple backpropagation filter will be created to remove undesired noise from an input signal. For this example we'll use a modulated high frequency signal contaminated by low frequency noise that we wish to eliminate. Traditional linear signal processing would employ a high-pass filter of the following form:

S(k+n/2) = F[x(k), x(k+1), ..., x(k+n)]

where S is the resulting filtered signal, k is the kth point in the time series and n is the number of samples being processing to compute the filtered point. F is the filter.

We'll model the backpropagation high-bandpass filter with the same inputs that are used in linear signal processing theory. We will use 11 input nodes (i.e., n = 11) to filter the contaminated signal and one output node to represent the filtered signal (analogous to S(k+5)). The input training data will consist of a contaminated signal and a "clean" signal. The signals have been formatted for Qnet input in the file FILT.DAT. A partial view of the contaminated signal is shown below. Use NetGraph to view the "clean" signal used for the target data.

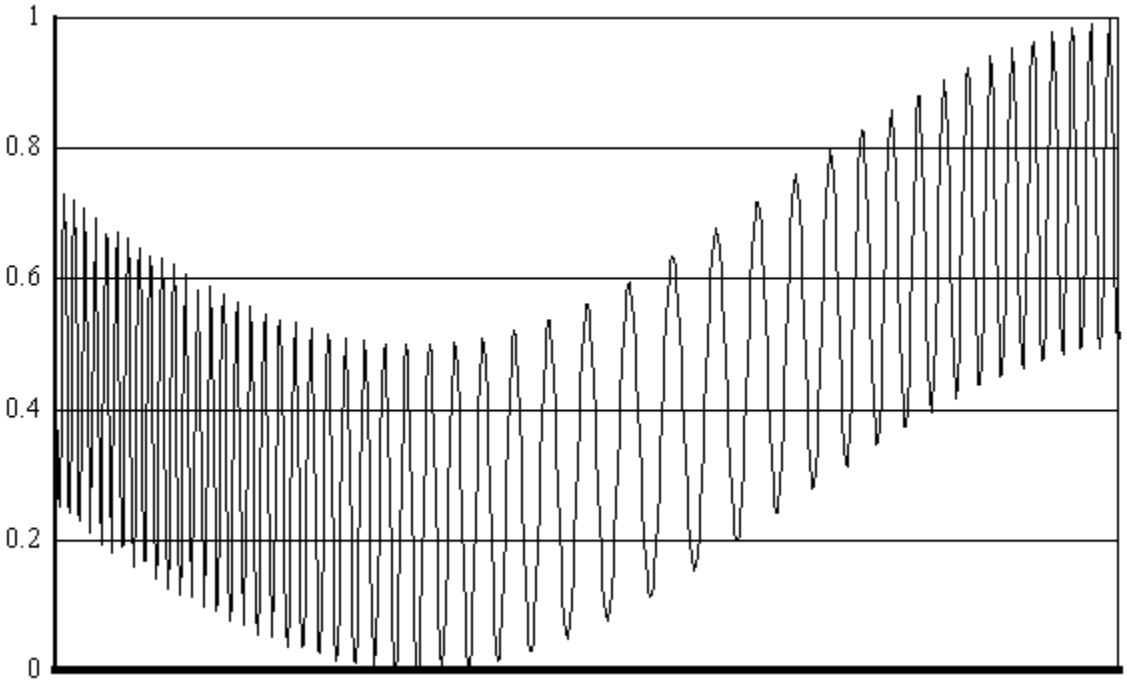
There are 1490 patterns contained in the input file. The training set consists of 1099 patterns and the overtraining test set uses 391 randomly selected patterns. Five layers are used in a fully connected network with 11 input nodes; 20, 15 and 10 hidden nodes; and 1 output node. The network employs the hyperbolic tangent transfer function.

The network files ***FILT.NET*** and ***FILTT.NET*** are available for examination. The ***FILTT.NET*** file contains a trained network and ***FILT.NET*** contains the untrained network definition. First, run the converged network and see the excellent job the network does filtering the larger amplitude low frequency noise from the desired high frequency signal. Qnet's graphing and statistical analysis tools show the excellent agreement for both training and test sets. The training was concluded prior to finding a global minimum in the test set error. It is likely that one does not exist for this example.

Try training the untrained network. Plot, adjust and control the training parameters as needed to improve training and convergence. Since the network is fairly large and uses a moderately large training set, overnight training may be necessary to reach the level of convergence shown in the trained example.

Backpropagation filters have been shown to be an excellent alternative to traditional digital filters. Currently, much theoretical and experimental investigation is taking place to determine the benefits of nonlinear signal processing offered by backpropagation neural networks.

Input Signal for Filtering:


1
0.8
0.6
0.4
0.2
0

Data Analysis/Sphere Drag

Example Problem:

Network Name: Sphere Drag
Number of Layers: 6
Connections: FULL
Input Layer:
- **Nodes:** 2
- **Transfer Function:** Linear

Hidden Layer 1:
- **Nodes:** 10
- **Transfer Function:** Tanh

Hidden Layer 2:
- **Nodes:** 8
- **Transfer Function:** Sigmoid

Hidden Layer 3:
- **Nodes:** 6
- **Transfer Function:** Gaussian

Hidden Layer 4:
- **Nodes:** 2
- **Transfer Function:** Sech

Output Layer:
- **Nodes:** 1
- **Transfer Function:** Sigmoid

This is an example designed to show how the scientist, researcher or engineer can use neural networks to analyze and model research data. A scientist gathers experimental data for some process or phenomenon in an attempt to predict and understand its behavior. It is also common that the theoretical model available to predict the phenomenon is either very inaccurate, extremely complex or simply not available. Traditional methods of using the experimental data to later predict a given result involves employing slow, complex multi-variable, table lookup techniques combined with various interpolation and data fitting schemes. These traditional techniques often fail when confronted with nonlinear behavior, data discontinuities, and non-uniform data sampling. Neural networks can easily handle such issues and offer reduced modeling time, increased prediction accuracy and reduced overall data storage requirements.

For this example, a model is needed to predict the aerodynamic drag on a sphere for a given range of speeds and atmospheric conditions. Let's assume the aerodynamic drag of a sphere is needed for air speeds ranging from Mach .3 to Mach 4.5 and for Reynolds numbers from 200000 to 600000. From fluid dynamics theory it is known that these are the two critical factors that influence aerodynamic drag. The researcher runs a series of wind tunnel tests to gather the data. For a sphere, the experimental drag data contains peaks, valleys and discontinuities (due to the onset of turbulence at the critical Reynolds numbers and Mach number effects). Using a neural network to model and predict sphere drag will allow the researcher to accurately capture these irregular features and utilize experimental data taken at non-uniform test conditions.

The example network is contained in ***SPHERE.NET*** (the untrained network) and ***SPHERET.NET*** (the trained network). The network model has 6 fully connected layers utilizing multiple transfer function types. The input layer has 2 nodes, one for the Reynolds number (divided by 10^5) and the other for the Mach number. The 4 hidden layers contain 10 nodes, 8 nodes, 6 nodes and 2 nodes respectively. The output layer has 1 node that represents the aerodynamic drag on the sphere. A total of six layers were used in this network in an attempt to better model the nonlinear behavior of the experimental data. We have 102 experimental cases to train the network. Instead of incorporating a test set, the resulting network will be interrogated by passing a large number of test cases through the network to visually check the results.

The results obtained from the network are excellent. The sphere drag data is accurately predicted for the training cases. The average error of the fit is less than 1%. The figure shown below is the result of passing 378 Mach/Reynolds number conditions through the network in recall mode. The resulting plotted surface models the drag trends extremely well, including the discontinuity that represents the onset of turbulence. Analyze the trained network by viewing the target/network output comparisons with NetGraph. Converge the untrained network to gain better insight into the convergence process. You will notice that it takes this network several hundred iterations get organized and start training, however, a sigmoid version

of this network took over thousand to get started. Adding gaussian layers to this network considerable improved convergence time.

Neural networks provide the perfect tool to model complicated research data. Creating a neural model will provide the researcher with an fast, accurate prediction tool for utilizing experimental data in research and design activities.

Data Analysis/Weather

Example Problem:

Network Name: USA Temperature Map
Number of Layers: 5
Connections: FULL
Input Layer:
- **Nodes:** 2
- **Transfer Function:** Linear

Hidden Layer 1:
- **Nodes:** 9
- **Transfer Function:** Sigmoid

Hidden Layer 2:
- **Nodes:** 7
- **Transfer Function:** Sigmoid

Hidden Layer 3:
- **Nodes:** 4
- **Transfer Function:** Gaussian

Output Layer:
- **Nodes:** 1
- **Transfer Function:** Sigmoid

In another data analysis example, meteorologists can use neural networks to model weather conditions. Conditions such as temperature, pressure or precipitation are often fed to the meteorologists from random locations over large geographical areas. Meteorologists must convert this scattered information into maps that can be used to convey the entire trend. Neural networks can be extremely effective at processing this data into uniform contour data for mapping. A neural network's ability to develop a complex, nonlinear data fit from non-uniform, scattered sources make Qnet the perfect tool for this type meteorological analysis. With faster PC hardware and an automated scheme for creating input training sets, the complete process of generating maps can be reduced to a matter of minutes. As the ability to generate sophisticated contour maps becomes easier, entire motion pictures of past weather histories and future predictions can be simulated with minimal effort. Beyond this practical application of neural network data analysis, meteorological researchers are exploring the use of neural networks in advanced weather forecasting models.

Our neural network example will be a simple temperature data fitting model. Temperature data will be modeled from various locations across the country. Our input nodes will contain the latitude and longitude of the cities reporting the temperature. The single output target will be the temperature. Once trained, the neural network will take an input latitude, longitude pair and produce a temperature. The model will then be used in recall mode to generate high fidelity temperature information from uniform latitude, longitude pairs. The results can be processed through a contouring/graphing application to easily produce our map data.

The example neural network is contained in ***TEMPS.NET*** (the untrained network) and ***TEMPST.NET*** (the trained network). The network model has 5 fully connected layers using sigmoid transfer functions. The input layer has 2 nodes. The first for latitude and the second for longitude. The 3 hidden layers contain 6 nodes, 5 nodes, and 4 nodes respectively. The output layer has 1 node that represents the target temperature. A five layer design offers both a sophisticated data fitting model and a short training time. The training data contains 233 temperature reporting cities from across the country. No test set is used since the resulting fit can be visually checked with the final map.

Results are very good considering the simplicity of our model. We did find that the response error grows unacceptably high in mountain areas. A better model for temperature mapping (but beyond the scope of this example) would be to include a third input node for altitude. With temperature being a strong function of altitude, we would improve the ability of the model to generate accurate temperatures in all areas of the country. Other than this exception, temperature trends are accurately captured. The temperature map shows the results of passing 1344 latitude/longitude pairs through the network in recall mode. The model's outputs were passed through a contour mapping application to generate the final picture.

Random Number Memorization

Example Problem:

Network Name:	**RANDOM MAPPER**
Number of Layers:	**3**
Connections:	**FULL**
Input Layer:	
Nodes:	**1000**
Transfer Function:	**Linear**
Hidden Layer 1:	
Nodes:	**3**
Transfer Function:	**Sigmoid**
Output Layer:	
Nodes:	**2**
Transfer Function:	**Sigmoid**

To show the use and possible misuse of neural networks we'll present a problem where we introduce 1000 random numbers to a network in an attempt to predict 2 random numbers. Obviously, one set of truly random numbers cannot be used to predict another set of random numbers. A neural network, however, can memorize the training set and map the input random numbers to the random outputs for the patterns presented to it (assuming the number of training patterns is small relative to the network size). To monitor network learning a test set will be used. If the model behaves as expected, it will memorize the training set cases, but make no learning progress with the test set.

This example uses a set of 8 training patterns and 4 test patterns. Each pattern contains 1000 random values for inputs and 2 random values for output (all normalized between 0. and 1.). The network is fully connected with three layers containing 1000 input nodes, 3 hidden nodes and 2 output nodes. The training data is contained in the file ***RANDOM.DAT***. Two network input files ***RANDOM.NET*** and ***RANDOMT.NET*** are available for examination. The file ***RANDOMT.NET*** contains a network where training has taken place and ***RANDOM.NET*** contains the untrained network definition.

The trained network easily mapped the input random numbers to the output random numbers. The network memorized the training set for the 8 training cases presented to it. Once the trained network sees the 1000 input numbers it can produce 2 output random numbers. This memorization, however, produces no constructive learning. The relationships developed during training are valid only for the training set . For problems where all possible cases are contained in the training set, memorization is fine and a productive neural network can be produced. Problems that require some type of predictive capability do not benefit from training set memorization. Incorporating a test set is the only way to monitor the training results. One factor that determines whether a network will memorize information is the relative size of the network versus the number of training patterns. For a case like this, with only 8 training patterns, memorization of the training set becomes likely. If we would have used 500 input training patterns, it is doubtful that significant memorization would have taken place (unless the network size were increased substantially).

Stock Forecaster

Network Name: Intel Predictor
Number of Layers: 4
Connections: FULL
Input Layer:
Nodes: 21
Transfer Function: Linear
Hidden Layer 1:
Nodes: 5
Transfer Function: Sigmoid
Hidden Layer 2:
Nodes: 2
Transfer Function: Sigmoid
Output Layer:
Nodes: 1
Transfer Function: Sigmoid

This example investigates the viability of using neural networks for developing a financial market forecasting tool. Academic arguments are often waged as to whether the stock market or individual stocks exhibit a "random walk" with virtually unpredictable trends or whether prices can be followed and forecast in the pursuit of profit. We will create a stock forecasting model to help us investigate this question. We will use historical stock prices (and technical indicators derived from the price/volume data) to predict future behavior. If the stock price is a true "random walk" we can expect to see results similar to the "RANDOM" example. For that example, the neural model was able to memorize the training set, but it could not respond accurately to inputs that were not part of the training set. If certain historical trends can be used to predict future prices, the neural model should be able to (at least partially) improve its response to test set cases, unlike the "RANDOM" sample problem.

Our sample stock forecasting model will be developed using Intel Corp.'s stock price. We will compute a series of our favorite technical indicators using the stock's High/Low/Close/Volume data over a five year period (1988-1993). A total of 21 indicators are computed for each day using the current and past days' trading information. The percent change in the next day's stock price is what we will predict with our model. Our desire is to develop a short term trading model that will indicate whether we should take long or short positions in Intel's stock.

Output node:

Column 1: Next day's close percent price change.

Input nodes:

Column 2: Current day's percent change from previous day's close.

Column 3: Current day's percent change from 2 days previous.

Column 4: Current day's percent change from 3 days previous.

Column 5: Closing ratio - where close is between daily high and low. (close - low)/(high - low).

Column 6: Normalized trading range - (high - low)/close.

Column 7: Normalized 3 day price slope (normalized by 3 day average).

Column 8: Normalized 8 day price slope (normalized by 3 day average).

Column 9: 3 day slope of column 7 (i.e. 2nd derivative of price).

Column 10: 8 day slope of column 8 (i.e. 2nd derivative of price).

Column 11: Price exponential moving average oscillator (8 day/4 day).

Column 12: 3 day slope of column 11.

Column 13: 5 day standard deviation of price / 5 day average price.

Column 14: 5 day standard deviation of volume / 5 day average volume.

Column 15: Close ratio - same as column 5 but with col. 13 standard deviation bands.

Column 16: Normalized on balance volume 3 day slope.

Column 17: Normalized on balance volume oscillator (8 day - 4 day exp. moving avg.)

Column 18: 3 day slope of column 17

Column 19: Normalized negative volume indicator (5 day exp. mov. avg).

Column 20: 4 day slope of column 19.

Column 21: Normalized positive volume indicator (5 day exp. mov. avg.).

Column 22: 4 day slope of column 21.

It is very important that all inputs be normalized or confined to a *bounded* range. Note that while the stock price and daily trading volumes for Intel have changed considerably over this 5 year historic period (these are unbounded values), the indicators (inputs) developed for the model produce *like values* regardless of the time frame observed. By scanning through the input node plots for this model, one can quickly see that each input has been normalized or confined within a standard range. The network's output is also normalized by predicting a percent change in price instead of the actual market price. Predicting the actual price or using unbounded inputs would seriously impair the effectiveness of this model. As a final data preparation process we limited all data to reside with 3 standard deviations of the median value. This is done to prevent *outlier values* from having a large impact on the model.

The network design contains 4 fully connected layers and utilizes the sigmoid transfer function. The input layer consists of 21 nodes that utilize the above indicators. The two hidden layers contain five and two nodes respectively. The output layer has one node for predicting the next day's close. A total of 1072 cases are used for training and 201 cases are incorporated into the test set and selected on a random basis.

The network definition files are ***INTEL.NET*** (the untrained network) and ***INTELT.NET*** (the trained network). The training data is in ***INTEL.CSV***. Training for this network will normally take a few thousand iterations. Errors decend rapidly after the initial flat startup period. After a relatively short learning period overtraining takes over, requiring the network to be returned to the optimal condition via AutoSave. However, with both training and test sets showing a decline in RMS error, it's evident that the price behavior is not completely random and that the current price behavior characteristics can be useful in forecasting future prices.

At first look the model seems to show only mild learning. For example, the correlation coefficient between the network predictions and training targets is only around .14 (high correlation is 1./no correlation being 0.) However, using Qnet's threshold checking, we see that a valuable trading model has indeed been developed. Both training and test sets show that the price forecasts generally increase in accuracy as threshold limits (the trigger or level that is used to initiate a buy or sell signal) increase in size. One would expect the model to be more accurate with its up or down prediction as the size of the predicted move grows. The included plots show this trend.

Our next step was to check the model using 1994/95 Intel stock prices (not part of the training set). We found that while the prediction accuracy was slightly below the levels obtained during the training process, the model still produced profitable results. Unfortunately, the commissions resulting from taking the many daily positions (either in the stock or underlying options) would wipe out much of the gains. Because of this, we must determine if a profitable trading scheme exists that allows one to hold positions over a longer term. Thoroughly analyzing the model's predictions over the 94/95 year, we found the model correctly generated strong buy and sell signals for virtually all intermediate term highs and lows. Using the model's predictions and these basic trading rules:

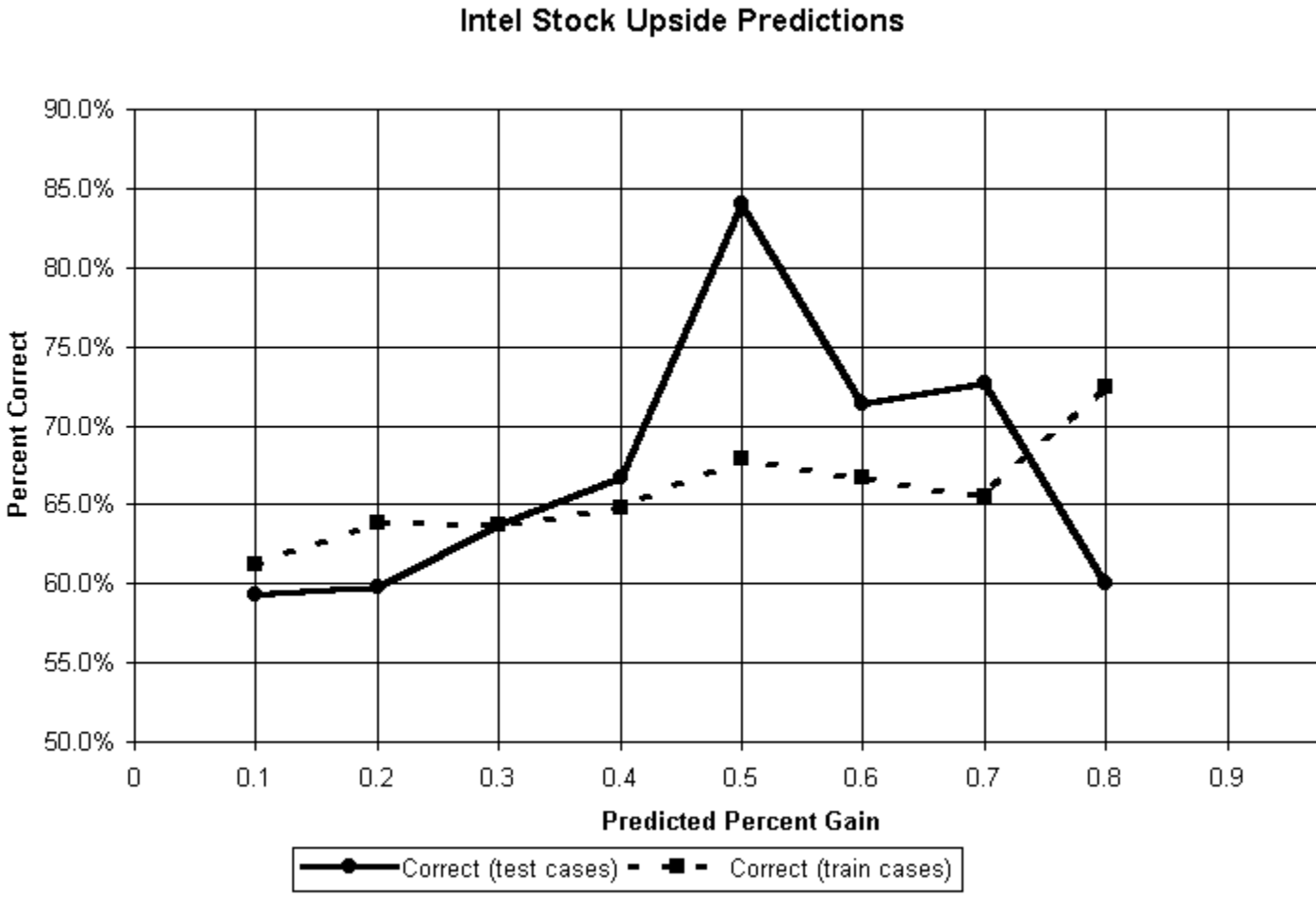
- ***Add a long position whenever a daily prediction of +.75 is exceeded.***
- ***Unload all long positions when a daily prediction of -.3 is exceeded to the down side.***
- ***Add a short position whenever a daily prediction of -.5 is exceeded to the down side.***

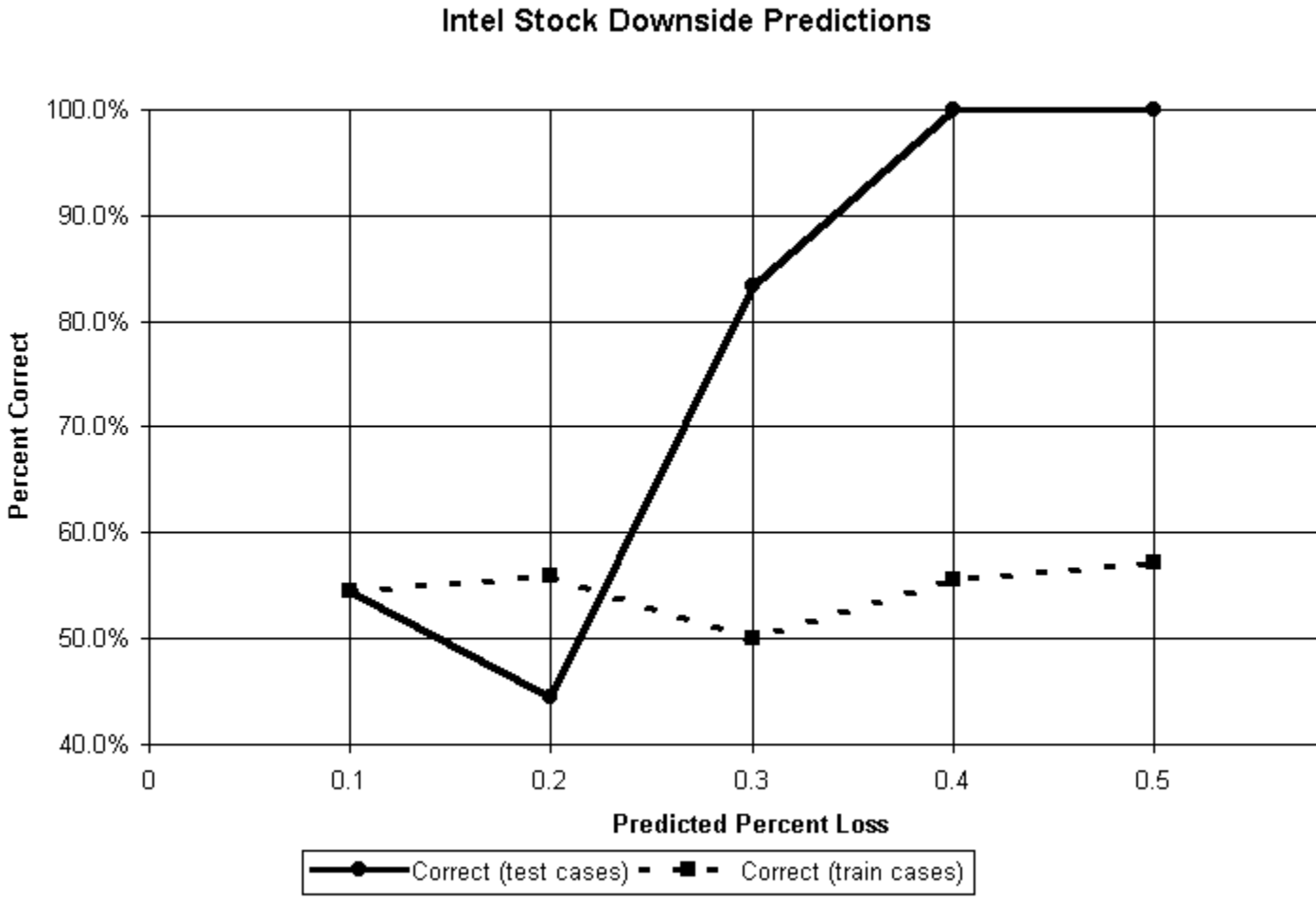
- ***Unload all short positions when a daily prediction of +.5 is exceeded.***

outstanding results are realized over this period. Following these rules, eight long positions and two short positions would have been taken during the year. The logic behind this set of rules is: ***Whenever the model generates a strong daily indicator number, the market over the short to intermediate term should move in that indicated direction. Whenever a position is being held and a moderate reading to the reverse side is generated, that market exposure should be eliminated***. An investor would have

safely been on the correct side of each major move ***prior*** to the move and no losing trades were experienced. Using 1 *"in the money"* option as a position, about $7500 in profits would have been generated. Obviously, both profits and risks can be increased by increasing the size that a long or short position represents.

This example shows the two step process required to build a trading model. First, you must develop the inputs, optimize the network design and produce a trained network. If positive results are seen, a trading scheme must be established that will optimize profits. To develop the above rules, we used a simple spreadsheet to analyze the model's prediction characteristics over an additional year of data. Advanced techniques that utilize genetic algorithms are also becoming popular for developing trading rules. Neural networks, when properly applied, can create invaluable price forecasting models that can be the basis for profitable equity, commodity or option trading schemes.




Intel Stock Downside Predictions
100.0%
90.0%
80.0%
70.0%
60.0%
50.0%
40.0%
Percent Correct
0
0.1
0.2
0.3
0.4
0.5
Predicted Percent Loss
Correct (test cases)
Correct (train cases)

Test Set Correlation History vs Iteration

Plot the run's test set correlation history. The correlation coefficient measures the how well the network predictions trend with the test set targets. This plot can be used in conjunction with test set RMS error and the test set tolerance history to help identify the onset of overtraining.

Test Set Tolerance History vs Iteration

Plot the run's test set tolerance history. The percentage of network predictions that fall within the user defined tolerance is plotted. This plot can be used in conjunction with test set RMS error and the test set correlation history to help identify the onset of overtraining.

Color Contours

Select any two inputs and generate color contours for a selected output. Qnet varies each input to produce a network output and displays the result in a color contour format. This can be used to find how input trends interact with output predictions. During the analysis, Qnet varies each input independently and, thus, validity of the output depends on the accuracy of this assumption. For best color contour results, use 16-bit or higher video modes.