Extensible Neural Network Software: Applications in Gene Expression Analysis

by

Jonathan Lee Jackson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author .....
Department of Electrical Engineering and Computer Science
September 17, 2004

Certified by .....
Lucila Ohno-Machado
HST Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Extensible Neural Network Software: Applications in Gene Expression Analysis

by

Jonathan Lee Jackson

Submitted to the Department of Electrical Engineering and Computer Science
on September 17, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Artificial Neural Networks have been increasingly utilized in the life sciences for analysis of large data sets. High-throughput technologies, such as gene expression microarrays, have challenged traditional statistical learning algorithms given their high dimensionality. This thesis describes GAINN, a neural network software package I created. GAINN was designed to be an extensible tool for both researches and students to use in neural network explorations. Several algorithms and features were implemented and tested on classification of various gene expression array data sets. The code design and user interface were implemented in such a manner that new algorithms and features would be trivial to incorporate into GAINN.

Thesis Supervisor: Lucila Ohno-Machado
Title: HST Associate Professor

Acknowledgments

I would like to thank my advisor, Dr. Lucila Ohno-Machado for her help and guidance throughout the entire process of completing this work. I would also like to thank Aaron Fernandes, for editing my writing which can barely be described as part of the English language at times. In addition, I would like to thank Nathan Vantzelfde for his help on user-testing and design ideas.

I would also like to thank the Department of Homeland Security Office of Science and Technology for their funding of my fellowship during the time that I completed this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the explosion of technology in the 20th century, the artificial intelligence community has seen many ideas arise with high expectations only to leave us with disappointment. Artificial neural networks[1] appeared as if they might follow this pattern in the late 1980s, when little practical application of neural network research was realized after an initial surge of enthusiasm. After a tumultuous start, neural networks have proven to be extremely useful tools for classification. From speech recognition, to character recognition, to missile defense systems, neural networks are well suited for a variety of classification tasks.

Here, I focus on their use for classifying gene expression microarray data sets. Gene expression microarrays using cDNA have only recently become available and allow us to measure thousands of gene expression levels with one sample [8]. This wealth of new data has drawn attention and study in classification of various diseases and conditions. As the data sets start to include a larger number of cases, microarray data become better suited for classification tasks. A few reports describe the potential of gene expression levels measured by microarrays in diagnostic and prognostic tasks [3, 11]. With microarray data, there exists a problem of determining which features are relevant to the classification task at hand. Due to the high dimensionality and relatively small number of data points, we must be careful to not over-fit the dataset

[1]Henceforth referred to as neural networks, with the exception of the section on the biological background of neural networks.

during training. For example, a leukemia dataset collected by Todd Golub, has 72 samples and 7129 features [9][2]. Neural networks with feature selection are well suited to take this vast number features and prune them down to a small subset. Many researchers have had excellent success using neural network for microarray classification [16, 24, 10]

As stated, the number of features provided with microarrays is extremely large, in the high thousands or more, and the number of training items is often less than 100. A majority of these genes are thought to be irrelevant for any one particular disease, and the number of features relative to the size of the dataset could create massive over-fitting. This creates two problems: how to best train the network given the structure and how to best choose that structure. My neural network package, GAINN[3], was created to address both of these problems. I explore training algorithms including back-propagation and its parameters: momentum, early stopping, and regularization. GAINN also uses a variety of cost functions to allow for different types of errors to be minimized.

1.1 Outline

The following chapter provides a brief introduction to gene expression data and why they are useful. Chapter 3 provides an exploration into the background of neural networks as well as the different components of neural networks that I included in GAINN. This includes a basic description of feed forward neural networks in general, and various descriptions of formulas that are used in GAINN. Chapter 4 discusses the GAINN software itself: how it was designed and how it is used. This chapter will explain my motivations for architecting the software in the way that I did. Chapter 5 examines GAINN's performance on gene expression array datasets. I evaluate the performance of GAINN against a small number of datasets provided in the public domain and discuss the results. Chapter 6 discusses the overall contributions of

[2]The original publishing states that there are 6781 genes measure. The dataset I used, obtained at, http://134.174.53.82/microarray, contains 7129.

[3]General Algorithms in Neural Networks.

GAINN and suggests future functionality.

Chapter 2

Gene Expression Microarray Data

The ability to obtain gene expression levels for a large number of genes only became available relatively recently. Experiments were performed at Stanford in 1995 on a small number of genes, and the first database of gene expression data was available in 1997[1]. Since then, gene expression datasets have become readily available and the target of a great deal of research, and there are now over many publicly available repositories of gene expression datasets.

The technique of using cDNA was developed by Schena in 1995 [32]. Microarraying is useful for studying the expression patterns of large numbers of genes. In this technique, single stranded pieces of known cDNA probes are attached to a glass slide or silicon chip. Purified mRNA from the sample is then run over this chip and expression levels may be determined by examining how much mRNA hybridized to the probes on the chip. Using this technique, several thousand genes may be measured at one time. The only requirement for a gene to be measured is that its signature is known before the experiment.

Gene expression levels can be used for a variety of purposes. One of the commercial uses with the highest potential today is in drug discovery. The under or over expression of a gene can be highly correlated with a disease. We expect that in some of these cases, the relationship is causal, and suppressing the over expression my help

[1]Henceforth, it is assumed that the term "microarray data" is referring to data obtained by gene expression analysis.

prevent or treat the disease. Such a gene would be a prime target for a drug therapy. Without further physiological properties, however, we would be unable to determine whether the over or under expression was the cause or result of a particular disease. Thus, gene expression data would need to be coupled with further tests in order to determine a causal relationship between the gene expression level and a particular ailment.

There are cases in which we are not concerned with whether the gene expression level is the cause or effect of a disease. In the case of the AML/ALL dataset by Golub [9], the data were collected to determine whether there were potential subclasses of known types of leukemia that were associated with different prognoses. Since the leukemias were labeled into the well known categories of acute myelocitic and acute lymphocitic leukemias, the data set has been used also to demonstrate the use of classification algorithms in this type of data.

We hope that eventually we may be able to use machine learning techniques—neural networks in particular—to diagnosis and treat patients based on their gene expression data. As mentioned, we face the difficulty of not yet having an abundance of data, though the available corpus is growing rapidly. One of the reasons for this is the difficulty of setting up an experiment. It is not a simple a matter of getting a blood samples from a particular patient. The level of gene expression varies from tissue to tissue and location inside a single person. Thus, we may need a sample from a specific organ, area, or tumor. However, in the gene expression datasets that we do have, excellent classification has been reported for some diseases [9] [34].

The purpose of GAINN was to create a comprehensive neural network tool for use in classifying datasets such as gene expression data. Using GAINN, I will attempt to accurately predict the class (disease) of patients based on gene expression data. Although I test GAINN using gene expression datasets, it is in no way limited to use on only this type of data. Virtually any type of dataset can be learned and predicted using GAINN[2].

[2]Obviously, GAINN's performance will depend on the nature of the dataset.

Chapter 3

Background

In this chapter, I will briefly discuss the development and history artificial neural networks. In addition, I will as describe the features of neural networks that are included in GAINN. Several derivations draw heavily upon the work of both Bishop [1] and Ripley [28].

3.1 Biology of Neural Networks

Much of the design of artificial neural networks stems from the awe inspiring processing power of the human brain. Due to its parallel processing abilities, the brain is able to perform some tasks much faster and better than even modern super-computers. The roots of artificial neural networks are based on their biological counterpart, the neural networks of living organisms.

The basic component of a neural network—both biological and artificial—is the neuron. The relevant components of a neuron cell are the dendrites, a soma (or body) and an axon. The synapse, or connection, between two neurons is composed of the axon of one cell and the dendrite of another. Information is transferred between cells via neurotransmitters moving through the extracellular fluid separating the axon and dendrites. When the electrochemical input of the dendrites reaches a high enough level, an action potential, or electrical signal, travels from the soma down the axon. When the end of the axon is reached, neurotransmitters are released through the

cell membrane and emitted across the synapse, where they are received by the connected dendrites of other neurons. There are approximately 30 neurotransmitters currently known, some of which are inhibitory but most of which are excitatory. The chemical process of the action potential causes the firing time of a neuron to be a few milliseconds, which is many orders of magnitude slower than today's supercomputers.

Each neuron has between 1,000 and 10,000 synapses, and the brain is estimated to contain around 100 billion neurons [17]. Although the firing time of a neuron compared to a modern day processor is about seven orders of magnitude slower, the brain has three orders of magnitude more connections—connections in a computer being the wires between transistors that make up the circuits. Modern computers operate mostly in series. In contrast, the brain is believed to operate heavily in parallel. As evidence, consider face recognition: a human can recognize a face in less than 100 milliseconds. Therefore, if neurons only operated in series, only 100 neurons would be used to recognize a face. It is believed that the slow rate of computation is overcome by massive parallelism, as this task could not be performed with only 100 neurons (this example is known as the one hundred step program) [13]. Modern day nuclear imaging also shows brain activity occurring in parallel in response to a single input change.

The artificial neurons used here mirror their biological counterparts in many ways, but there are some differences. The artificial neurons in GAINN have single valued outputs. Biological neurons transmit via pulses of neurotransmitters, allowing for two more degrees of freedom during a firing: the phase of emissions and the frequency of emission pulses [19]. It is quite plausible, however, that these extra degrees of freedom do not represent any real information and are random variations based on the extracellular conditions. Therefore, treating the output as a single valued output is at least appropriate and may be equivalent to its biological counterpart. These variations, random or not, cause the output of a neuron given the same input may have a possibly different effect on the neighboring neurons. In GAINN, the same inputs to an artificial neuron will always produce the same output.

3.2 History of Artificial Neural Networks

The earliest artificial neuron design was published by Pitts and McCulloch in 1943 [20]. In their research, they discussed the idea of "threshold logic." The basic design of their artificial neuron, or perceptron, takes in a set of binary valued inputs and pulses an output if this sum is greater than a certain threshold. In a perceptron where there are two inputs and the threshold is 1, this would correlate to an OR logic gate in Boolean logic. The inputs may also have certain weights associated with them, thereby allowing certain inputs to weigh in more or less on the total summation.

In 1949, Hebb published a paper suggesting that if two neuron cells, A and B, regularly fire together, then some change will occur in the brain's chemistry that will facilitate the firing of A when B is also firing [12]. That is, they will now be more likely to fire together again; it was thought that this could be how the biological brain "learns." This idea was put to use in 1954, when Farely and Clark created artificial neural networks based on Hebb's idea [6]. This network required a large number of artificial neurons and its structure was predetermined. This early design was able to discriminate between two different patterns if they were dissimilar enough and fed in the correct order.

In 1958, Rosenblatt published work on a new perceptron in the his appropriately named book, *Perceptron* [29]. His perceptron had three layered parts: sensory, association, and response. This model continued to shift the focus from a randomly connected neural structure to a predefined structure. The difficulty then moved from determining the network structure, to determining what the weights of the predefined structure should be. In 1962, he published a learning algorithm demonstrating that his model was able to categorize linearly separable patterns into two different groups [30]. However, in some cases, it would only perform correct classification if the patterns were presented in the correct order. His work, unlike others at the time, did not focus on the structural connectivity of the neural network.

The first commercial success of neural networks was enjoyed by Widrow and Hoff of Stanford. They created ADALINE and MADALINE which were named for their

use of ADAptive LINear Elements; MADALINE was simply multiple ADALINEs. This model used a linear activation, rather than the threshold activation used in previous models, which more closely mirrors its biological counterparts. In 1962, Widrow published a rule for use in training based on Least Mean Square (LMS) error minimization [36] [1]. Rosenblatt then merged this work with that of ADALINE to give rise to a new perceptron, which is the forefather of those used today. Before ADALINE was created, the input lines were only multiplied by weights after the summation unit. ADALINE move the weights before the summation unit so that each individual input line could be weighted. MADALINE was used commercially to minimize echoes on phone lines and is still in use today.

The success of MADALINE and the work of Rosenblatt generated a great deal of interest and expectation for the neural network field that was beyond reach of their performance at the time. In 1967, Minsky published a paper stating that neural networks were capable of universal computation and analyzed them as finite state machines. However, just two years later, Minsky and Papport published a book discussing the shortcomings of the current perceptron model [21]. In particular, a single perceptron was incapable of separating nonlinear datasets—its failure on an XOR dataset being one of the classic illustrations[2]. This influence, along with failed expectations, greatly reduced research in the area for the next decade.

In 1982, research interest was renewed as John Hopfield of Caltech introduced Hopfield Nets, which contained a novel way to connect various perceptrons using bidirectional lines. The new use of bidirectional lines would eventually lead to the development of multilayered perceptron models, where the output of a summation node is the input to another summation node. The multilayered perceptron would become the basic structure of feed-forward artificial neural networks that is used today [14].

It was still unclear, however, how to train these multi-layered perceptron mod-

[1]Error functions will be discussed in detail in section 3.9.

[2]The XOR dataset is the smallest nonlinearly separable data set and consists of one class containing (1,0) and (0,1), and another class containing (1,1) and (0,0). There is no straight line that can be drawn on an XY plane that completely separates these classes.

els and some concern arose as to whether a training algorithm was even possible. Fourtunately, these concerns were put to rest in 1986 when Rumelhart, Hinton, and Williams published a paper detailing an algorithm based on the delta rule that could be used to train MLPs with hidden layers [31]. The algorithm, dubbed "Back Prop", involves propagating the output error backwards through all of the connections and is discussed in section 3.5. This algorithm was widely successful and coupled with the explosion in computing power helped to launch the extensive use of neural networks. Today, many successful companies employ neural networks in some capacity as predictive tools. A majority of neural network research is still based off of this simple algorithm and its variations.

3.3 Single-Layer Perceptrons

Single-layer perceptrons from the 1960s gave rise to the networks used in GAINN: feed forward multilayer perceptrons. These earlier single-layer perceptrons were created by Frank Rosenblatt based on research from McCulloch and Pitts, Hebb, and many other before him, including neurologists [20, 12].

The basic building blocks (artificial neurons) of these earlier networks differ from what most researches use today. Modern neurons have inputs connected directly to weights before reaching the threshold activator units—the equivalent of activator build up in a biological neuron. The 1960s single-layer perceptron, however, had the inputs connected to randomly assigned -1 or +1 coefficients and then fed into a threshold activation unit. The outputs of these threshold activation units were then connected to weights and fed into a summation unit as depicted in figure 3-1.

The input lines, or wires, are multiplied by +1 or -1[3]. The first layer of threshold units simply sums the value of these input lines. Then, a 1 is output if the sum is above a certain threshold and zero is output otherwise. The training—or learning—of the perceptron is performed by adjusting the weights (the round squares in figure 3-1). Based on these weight values, it was hoped that it would be possible to classify

[3] +1 and -1 coeffecients were used to produce different inputs into each threshold unit.

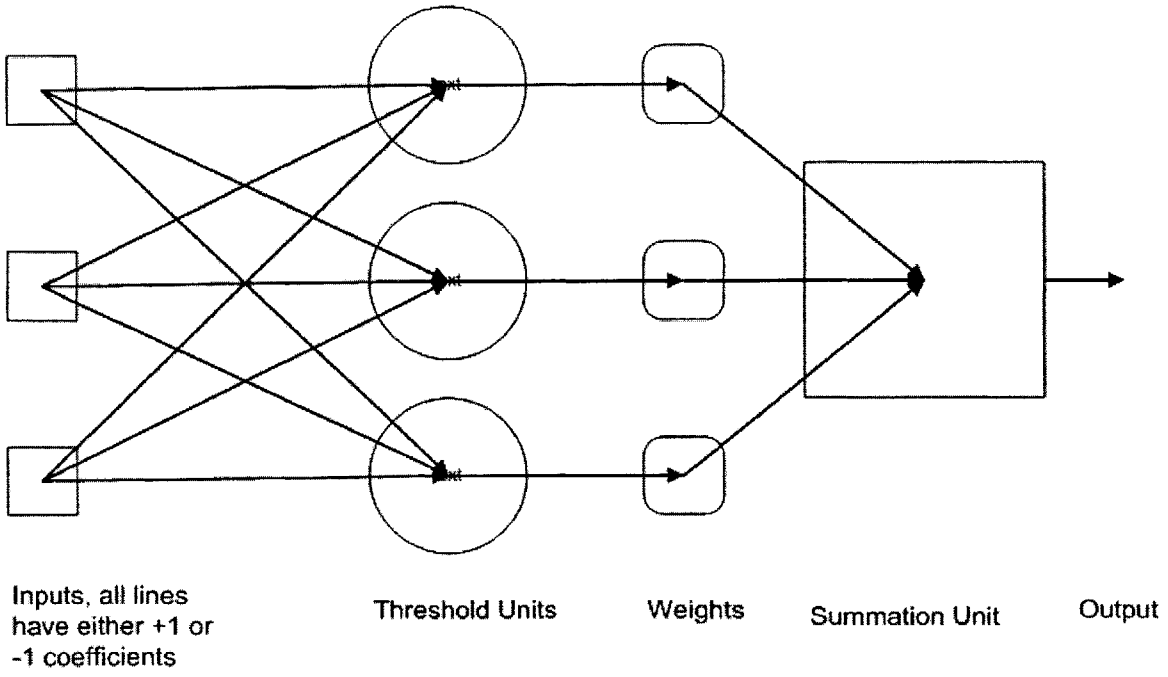

Figure 3-1: The early attempt at a perceptron.

different sets of inputs.

Rosenblatt proposed the following algorithm for training this perceptron [30]:

1. Send a training item through the network and examine the input.
2. If the output is correct, continue
3. Else, adjust the weights according to $a_k(n+1) = a_k(n) + y_k(n) * \varphi(n)$. Where $\varphi(n)$ is equal to 1 if the training item is in set 1, and -1 if the training item is in set 2, $y_k(n)$ is the output for the nth item of the kth unit, and $a_k(n)$ is the weight for the kth unit.

Rosenblatt proved that if the training data was linearly separable, the perceptron's weights would converge in such a matter that the training data could be classified without error.

Minksy and Papert modified the perceptron to remove the initial +1/-1 fixed weights and the initial threshold nodes. Thus, the input lines were connected directly to the adjustable weights which then fed into the summation unit. This model is

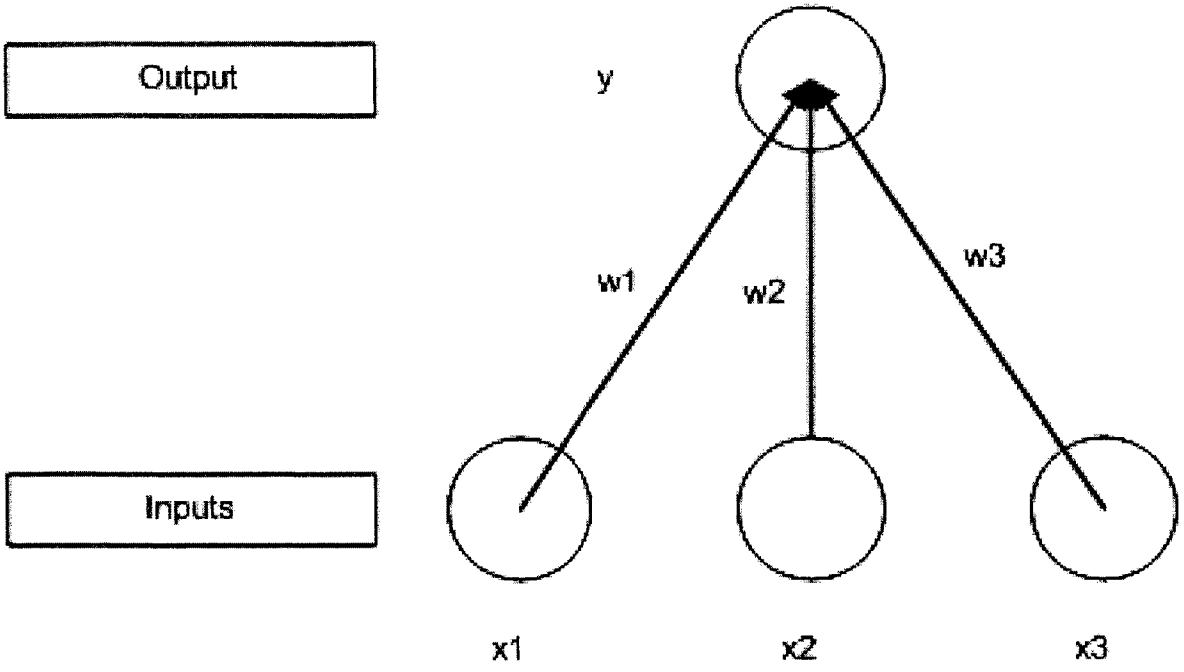

Figure 3-2: A single layer perceptron with one output.

known today as the single layer perceptron, and can classify any linearly separable dataset, figure 3-2.

The output of this perceptron can be expressed cleanly as:

$$\mathrm{y} = \sum_{\mathrm{i}=1}^{3} (\mathrm{w_i} \times x_i) + bias \tag{3.1}$$

The bias term is not shown in figure 3-2. However, we can eliminate the separate bias term if we add another input node x_0 that is always equal to one. Then the weight of this connection w_0 becomes equivalent to the bias. More generally, we can consider a single layer perceptron a linear discriminate function

$$y(\mathrm{x}) = g(\mathrm{w}^T \mathrm{x} + w_0) \tag{3.2}$$

where w is a vector of weights and $g(\cdot)$ is any function. If there were no bias term x_0 in the neural network, then the separating hyperplane would be forced to go through the origin in n-space. We would then no longer have the ability to separate any linearly separable dataset in a single layer perceptron.

There can also be multiple output units, each with its own set of weights, for use in categorization of multiple classes. Furthermore, the activation function of the output neuron need not be linear (a simple summation), it can also be logistic or other non-linear functions. These activations functions will be discussed further in section 3.5. Ultimately, however, regardless of the activation function, the single layer perceptron is only capable of separating linearly separable sets. Minksy and Papert prove this limitation as discussed in section 3.2. This severely diminished interest in the perceptron model until it was upgraded to the multilayer perceptron [21].

3.4 Multilayer Perceptron

The networks that will be used here are feed forward networks or *multilayer perceptrons*. These are comprised of neurons organized into layers such that all outputs of one layer can be computed and then fed into the next layer as depicted in figure 3-3. According to Ripley, a feed forward network is

> *A network in which vertices can be numbered so that all connections go from a vertex to one with a higher number. In practice the vertices are arranged in layers, with connections only to higher layers* [28, p. 349].

Because of this property, an explicit function can be written to express the outputs in terms of the inputs. And this function, depending on the activation function of the neurons, can be differentiable. Other types of networks do exist, such as *recurrent* networks. Since these networks can have connections that form loops, it is not possible to express the outputs of these networks as explicit functions of the inputs. In some cases, these networks even contain bidirectional connections between nodes [33].

Figure 3-3 depicts a basic feed forward network with one **hidden layer**. A hidden layer in an multi-layer perceptron (**MLP**) is a layer that is not an output layer or and input layer. The input to each unit's activation function is a_k:

$$a_k = \sum_{j=1}^{n} (w_{jk} \times input_j) + bias_k \tag{3.3}$$

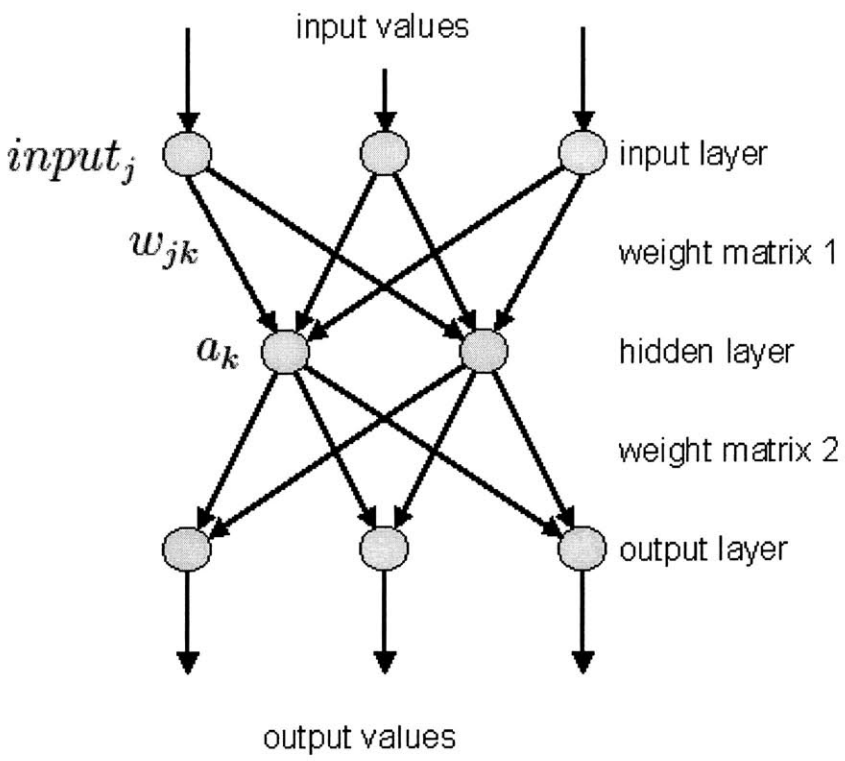

Figure 3-3: A basic feed forward network with hidden neurons.

Here, a_k is the input to the kth neuron, w_{jk} is the weight from the jth neuron to kth neuron, a_k is the value input to the activation function of the kth unit, and $bias_k$ is the bias of the kth unit. The output of the kth unit is then given as

$$z_k = g(a_k) \tag{3.4}$$

where $g(\cdot)$ is known as the activation function. There are multiple types of functions one could choose to use here:

$$g(a) := \begin{cases} 1 & \text{when} \quad a > 0 \\ 0 & \text{when} \quad a \leq 0. \end{cases} \tag{3.5}$$

$$g(a) = \frac{1}{1 + e^{-a}} \tag{3.6}$$

$$g(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{3.7}$$

In order, these are known as threshold, logistic, and hyperbolic tangent functions. While these three formulas look very different, a choice between them does not inherently limit the abilities of a neural network. The threshold activation certainly seems to be more limiting that the other two types of activation functions, and yet, a neural network with only 2 hidden layers using only threshold functions can approximate any function arbitrarily close [1, p. 124][4]. Furthermore the logistic function differs from the hyperbolic tangent function only by a linear transformation. Thus, a network trained using (3.6) could be turned into a network using (3.7) by transforming the weights and biases [1, p. 127].

This is not to say that different activations functions do not have their advantages. Threshold activations as compared to sigmoidal activations usually require more complicated networks with a great number of hidden units and layers to approximate the same function. And empirically, the hyperbolic tangent function has been shown to produce faster convergence than the logistic function[28].

Using any of the above activation functions, it is possible to calculate the output unit's value in closed form. For the network pictured in figure 3-3 with three inputs and two hidden neurons, the value of each output unit k can be calculated as follows:

$$a_k = g(\sum_{i=1}^{2} w_{ik} \times g((\sum_{j=1}^{3} (w_{ij} \times input_j)) + bias_i) + bias_k) \tag{3.8}$$

The output of this MLP can be written in closed form. In recurrent networks, it is not possible to simply differentiate the output with respect to the inputs, since feedback loops may make this impossible. However, if $g(\cdot)$ is differentiable, then (3.8) will also be differentiable with respect to the weights and inputs. The fact that the partial derivates can be calculated explicitly is the key to the back-propogation algorithm that will be discussed in section 3.5.

[4]When using threshold activations functions, however, the number of hidden neurons usually necessary to approximate a function is much larger than the number that is necessary using sigmoid activation function.[1]

The ability of the MLP to classify non-linearly separable datasets $g(\cdot)$ comes from the nonlinear activation function of the neurons. If $g(\cdot)$ were linear, we could remove any hidden units because a network with the inputs connected directly to the outputs would be equivalent, such as in a single layer perceptron [22].

There can be more than one hidden layer in the feed forward network. However, a network with enough hidden units in just one layer would be able to approximate any function as Radford Neal—an expert in the field of neural networks—writes [22]:

> Several people [5, 7, 15]have shown that a multilayer perceptron with one hidden layer can approximate any function defined on a compact domain arbitrarily closely, if sufficient numbers of hidden units are used. Nevertheless, more elaborate network architectures may have advantages, and are commonly used. Possibilities include using more layers of hidden units, providing direct connections from inputs to outputs, and using different activations functions. However, in "feedforward" networks such as I consider here, the connections never form cycles, in order that the values of the outputs can be computer in a single forward pass, in time proportional to the number of network parameters.

As mentioned above, however, when using threshold activation units, it is necessary to have two hidden layers to approximate any function. It has just been shown that neural networks should be able to classify many datasets. The question is, then, how we train the weights in the neural networks to separate these classes.

3.5 Back-Propagation Training Algorithm

The basic algorithm that is used in GAINN is known as error back-propagation. This algorithm was published by multiple authors at various times. The algorithm can be found published first in 1969 by Bryson and Ho and then again in 1974 by Werbos [2, 35]; however, both of these publishings went largely unnoticed. The algorithm became widely known in 1986 when three researches each published independently around the same time Rumelhart, Parker, and LeCun [31] [25] [18].

The basis of the algorithm is a gradient descent to modify the weight settings to minimize an error function. Error functions are not unique to neural networks and are used in almost all machine learning techniques. One of the most common error functions is sum-of-squares error,

$$Error = \frac{1}{2}\sum_{k=1}^{n}(y_k - t_k)^2 \tag{3.9}$$

where n is the number of training item, y_k is the output for a training item, and t_k is the desired output. We seek to minimize this error by adjusting the weights of each connection in the network. We can easily determine the error of an output neuron based on what its desired value was. From this, we can find the derivative of the error with respect to the inputs to an output unit. In the case of a single layer perceptron (no hidden units) minimizing the error function becomes a simple problem of optimizing the weight parameters of one layer and can be solved explicitly [1, p. 92].

However, in the case of multiple layers, we cannot explicitly determine which weights we should adjust in order to minimize this error. We do not have an equivalent desired output vector t_k for hidden neurons and thus cannot easily find the derivative of the error. Fortunately, we can "propagate" the error backward in order to determine the partial derivate of any weight with respect to the error.

For some arbitrary weight w_{ji} going from unit j to unit i, we can express the partial derivate using the chain rule as

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{3.10}$$

where E^n is the error on the nth training data item; remember from (3.3) that a_j is the input into the activation function after the summation of all of its inputs and bias. Since we are simply summing the weights times the inputs to find a_j,

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{3.11}$$

where z_i is the output of unit i that has a connection to unit j. Threfore, (3.10)

becomes

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j} z_i \tag{3.12}$$

We are now left with the difficulty of determining $\frac{\partial E^n}{\partial a_j}$, but this turns out to be easy. For output units, we have

$$\frac{\partial E^n}{\partial a_k} = g'(a_k) \frac{\partial E^n}{\partial y_k} \tag{3.13}$$

Note that the subscripts have been changed to denote this unit, k, as an output unit. In words, the above equation is simply stating that the derivative of the error function with respect to the input of output unit k is the derivative of the error function with respect to output y_k times the derivative of the activation function. For hidden neurons this equation becomes

$$\frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{3.14}$$

Hidden units can only influence the error by their output; since their output is only a function of their input, we can evaluate this function to

$$\frac{\partial E^n}{\partial a_j} = g'(a_j) \sum_k w_{kj} \frac{\partial E^n}{\partial a_k} \tag{3.15}$$

Since we can explicitly calculate $\frac{\partial E^n}{\partial a_k}$ because we know the output of the network, we can "back-propagate" the errors to the hidden layer. $\sum_k w_{kj} \frac{\partial E^n}{\partial a_k}$ can be thought of as the contribution of this hidden unit to the error of each of the output units. If we first computer the output layer's partial derivates, we are able to calculate all the derivates by moving backwards one layer at a time: this is where the term "back-propagation" arises from.

In the case of a neural network with a sigmoid activation function and a sum-of-squares error function, the derivative of the activation function (3.6) is

$$g'(a) = g(a)(1 - g(a)) \tag{3.16}$$

This has great appeal from a computer science perspective because it is easy and fast to calculate. The derivative of the error function (3.9) can also be easily computed:

$$\frac{\partial E^n}{\partial a_k} = y_k - t_k \tag{3.17}$$

In order to turn this into an algorithm, we must use these derivatives to adjust the weights for better classification. Two basic techniques for doing so are batch and online training. Batch training updates the weights after a run through the entire training set or **epoch**, such that

$$\Delta w_{ji} = -\eta \sum_n \frac{\partial E^n}{\partial a_j} x_i^n \tag{3.18}$$

while online training updates after each training item is evaluated

$$\Delta w_{ji} = -\eta \frac{\partial E^n}{\partial a_\mathrm{j}} x_i \tag{3.19}$$

The learning rate, η, is the speed at which we want to change the weights or move down the gradient. Setting the proper learning rate and/or modification of it during training is critical for efficient training of neural networks. If the learning rate is too high, the neural network will be slow to converge because it will frequently move too far on the surface and will move past the descent in the slope. If the learning rate is too slow, the neural network is more likely to get stuck in many local minima.

3.5.1 Momentum

A popular extension to basic back propagation is to use momentum [28, p. 154]. Momentum factors in the previous weight change in the current iteration so that the update of the weights becomes

$$\Delta w_{ji}^{n+1} = -\eta \frac{\partial E^n}{\partial a_\mathrm{j}} x_i + \alpha \Delta w_{ji}^n \tag{3.20}$$

where α is the amount of momentum to use. This smoothing allows for the algorithm to (hopefully) skip over small local minima it might otherwise get stuck in by continuing its general direction. This also tends to be more helpful with large learning rates, as the smoothing helps to prevent the algorithm from jumping around on the surface as it finds a local or global minimum.

3.5.2 Batch versus Online training

It is not obviously clear whether batch or online training should lead to faster convergence of a network. For a fixed learning rate, η, the batch algorithm can converge but the online version will wander for ever (although it may simply be wandering around the absolute minimum) [28, p. 155]. Theoretically the online algorithm should converge must faster if the dataset is large and repetitive, as a small sampling would provide the same effect as running through the entire dataset. This is not a worry in microarray data analysis as we are typically starved for data points to begin with. Online is also thought to be advantageous because the random sampling introduces noise that will hopefully allow it to avoid local minima [1, p. 264].

3.5.3 Stopping

With any iterative training algorithm, we must know when to stop. The danger in stopping too late is to overtrain the neural network. We have already shown that the multilayer perceptron is capable of approximating any function arbitrarily close. If we assume that our training algorithm will eventually converge, then we will be able to find a separating hyperplane for any dataset that is separable. However, if we let the training error go to an absolute minimum, then we will not be generalized well for use on a test set, even though the network performs perfectly on the training set. This is a common problem in all machine learning algorithms which do not have explicit solutions. The training error will always be decreasing throughout training; however, there will be a point at which the test error starts to increase. There are no known easy measures of where this point exists, though many techniques have been

suggested and some are implemented in GAINN.

3.6 Regularization

As just mentioned, we want to be careful not to overtrain these networks. One common technique for machine learners is to introduce a penalty in the cost or error function for the complexity of the system. Using this error function the neural network, in accordance with *Occham's Razor*, will favor a less complex system if it can classify just as well (or perhaps even worse) than a more complex system. In the case of neural networks, we can consider the size of the weights to be a measure of the complexity. Thus, if we modify the sum-of-squares error to have a penalty term

$$E' = E + \lambda C \tag{3.21}$$

where λ is the amount by which we want to factor in this penalization and C is some measure of complexity, we can now follow the back-propagation algorithm as before and regularize the weight settings to favor smaller weights.

The overall goal of regularization is to create a neural network that is more generalized; this will theoretically perform better on test data since it is not overtrained. Presumably, the smaller the weights, the smoother the function should be. By penalizing larger weights, we are forcing the back-propagation algorithm to have a trade-off between the smoothness of the surface and the total error. This will help to avoid over-training. One basic equation to use for C is

$$C = \frac{1}{2}\sum_i w_i^2 \tag{3.22}$$

When we use this as the error function and sigmoidal activation units, the weight updates in (3.19) get modified to

$$\Delta w_{ji} = -\eta \frac{\partial E}{a_j} x_i - \eta \lambda w_{ji} \tag{3.23}$$

This causes the weights to be smaller in general [28]. It is also possible to encourage

very small weights to decay faster by using an error function of

$$E' = E + \lambda \sum_{ij} \frac{w_{ij}^2}{1 + w_{ij}} \tag{3.24}$$

This is known as weight elimination rather than weight decay, because smaller weights will be decreased to near zero. The goal of both of these techniques is to cause the function that fits the data to be smoother.

3.7 Alternative Cost Functions

While sum-of-squares error is the most common and most basic, there are other cost functions that can be used and are implemented in GAINN. One of the problems with a simple sum-of-squares error cost function (with or without regularization) is that a single mislabeled data point or a single outlier will severely inhibit the neural networks ability to classify data. The sum-of-squares error measure will fit a line that is effectively the conditional mean of the data [1, p. 210]. One outlier could skew the mean by a large amount, especially in the case of microarray data where data points are scarce to begin with.

One alternative cost function is known as the Minkowski error,

$$E = \sum_{n} |y(x^n) - t^n|^R \tag{3.25}$$

where R is the can be chosen to be any value (this is also known as the Minkowski-R error). In the case where R = 2, we have our basic sum-of-squares error. When R = 1, we are left with an error function that will converge on the conditional median of the data, rather than the mean. Therefore, outliers and mislabeled data points will not have as drastic an effect on the separating hyperplane found by the neural network during training. In addition to this problem, the sum-of-squares error measure makes certain assumptions about the underlying model generating the dataset.

The sum-of-squares error is derived from a belief that the dataset was generated from a smooth deterministic function with Gaussian noise [1, p. 230]. In the case of

microarray data, however, it is plausible that the data were not generated from such a function. In fact, we are often only distinguishing between two different classes—such as types of cancer or positive and negative prognosis. Since the outputs are binary, 0 or 1, there is clearly no Gaussian noise in the output of the function. It makes sense, then, to look for an alternative cost function that is better at classifying this type of binary output; an equation that fits this criteria is known as the cross-entropy error measure

$$E = \sum_n \{t^n \ln(y^n) + (1 - t^n) \ln(1 - y^n)\} \tag{3.26}$$

This makes the derivative of the error function

$$\frac{\partial E}{\partial y^n} = \frac{(y^n - t^n)}{y^n(1 - y^n)} \tag{3.27}$$

and the partial derivate at the hidden layer (using sigmoid activation units)

$$\frac{\partial E}{\partial a^n} = y^n - t^n \tag{3.28}$$

This error measure, coupled with sigmoid activation functions, gives special meaning to the output value of a single neuron for two classes. The value of the single output neuron will be equal to the probability that the data item belongs to a particular class, conditioned on the input vector [1].

3.8 Validation and Ensemble Averaging

Now that we have established the training methods, it is necessary to determine how well the network is performing. Let us take Todd Golub's dataset, for example, which contains microarray data for 72 different patients, divided into two classes or cancer: acute myeloid leukemia (AML) and acute lymphoblastic leukemia (ALL). One approach is to just train the network until we reach an absolute minimum in the training error (for whatever error function we may choose). However, this will undoubtedly lead to an over-trained network and bad generalization on a test set

[1, 28, 13]. There would ideally be a point in the training error decent at which we could stop and know that this is the optimal place to stop training. However, not such metric or formula exists to determine where this stopping point is.

One solution is to use a hold-out set as a test-set on training. So, for example, we could divide the 72 patients into two groups of 36. Train the network on 36 of the patients, and then test the network on the other 36. While we are moving down the gradient of the training error, we continuously check the performance on the other 36 patients that we did *not* use for training. Both errors should stat to decrease at first, but then the validation error will begin to rise as the neural network overfits (or *loses generalization* on) the training set. We would stop at the point in which the validation error began to rise.

Another popular technique is called n-fold cross-validation. The test set is divided into n groups; so for n = 6, we would have groups of 12. The neural network is then trained on n-1 groups and tested on the nth group that was left out. After all groups have taken been left out, the network that performed the "best" is kept.

Both of these techniques have their downside, however. In the case of just using one hold-out set, we do not have the benefit of multiple random starting points. Since the error surface is rough and may contain deep local minima, the starting point could easily make the difference between getting stuck in a local minima and finding a true minimum. Also, the performance could be greatly affected by how the dataset is split. In the case of n-fold cross-validation, typically, only the best network is kept. Therefore, time is wasted training the n-1 networks that are not going to be kept. Also, the network that performs best is likely performing best because of a favorable data split. For example, if the 12 patients in one group of the 6-fold cross-validation were all extremely easy to classify, the network trained with this group as the validation set would perform better even if it was not the best overall network when tested using a hold-out set. Because the subsets are combined to form the training data, there is no way to determine the "best" network based on the validation results [23].

One solution to this problem is to use an ensemble of neural networks. Once

again, this technique is not unique to just neural networks but is also used in many other machine learning techniques. Presumably, there has been some computational time spent training multiple networks; it would be better if we could use all of these networks for classification rather than just the "best" one. I use the word "best" loosely here, because there is no way to ensure that any one network is in fact the best. The basic idea of an ensemble network (or committee) is to use all of the trained networks to each contribute to the output of the network for a given input [26]. That is,

$$y(x) = \frac{1}{n}\sum_{i=1}^{n} y_i(x) \tag{3.29}$$

where $n = 6$ in the above example; in this case, each network contributes equally to the output. If we assume the networks output $y(x)$ is equal to the actual function that generated the data, $h(x)$ plus some zero mean Gaussian error, we have:

$$y(x) = h(x) + e(x) \tag{3.30}$$

The expectation of the squared error of network i is

$$E[error_i] = E[\{y_i(x) - h(x)\}^2] \tag{3.31}$$

The average squared error for the committee is

$$E[error] = E[(\frac{1}{n}\sum_{i=1}^{n} y_i(x) - h(x))^2] \tag{3.32}$$

Now, if it is assumed that the errors are independent, then many terms drop out of the above equation when it is expanded. Based on our assumptions of Gaussian distribution and the independence of errors,

$$E[error_i] = 0 \tag{3.33}$$

and

$$E[error_i \times error_j] = 0 \tag{3.34}$$

when $i \neq j$. This means that the expected error for the ensemble output becomes

$$Error_{ens} = \frac{1}{n^2} \sum_{i=1}^{n} E[\{y(x) - h(x)\}^2] \tag{3.35}$$

as compared to the expected error of the average of the networks

$$Error_{avg} = \frac{1}{n} \sum_{i=1}^{n} E[\{y(x) - h(x)\}^2] \tag{3.36}$$

Combining (3.35) and (3.36) we see that the error of the ensemble is $\frac{1}{n}$ the error of the average network, which is a dramatic improvement. However, there are two problems with the assumptions made in this derivation. First, we cannot assume that the output of each network equals the real function plus some small amount of noise. Second, the errors of the two networks are almost definitely not independent. Since the network are trained using n-fold cross validation, the training patterns overlap a great deal and the correlation of the errors is high. Therefore, the gain over the average error is much smaller than $\frac{1}{n}$ but no worse than the average error [1].

3.8.1 Optimal Voting Weight

The above derivation assumed equal weighting of each network in the ensemble. However, there is a better way to weight the networks. The networks are not going to behave equally well, and it makes sense to weight some of the networks *vote* more than others. The optimal weights for the networks to minimize the error is

$$\alpha_i = \frac{\sum_{j=1}^{N} (C^{-1})_{ij}}{\sum_{k=1}^{N} \sum_{j=1}^{N} (C^{-1})_{kj}} \tag{3.37}$$

where C is the correlation matrix. The approximation for each element in the correlation matrix is:

$$C_{ij} \simeq \frac{1}{N} \sum_{n=1}^{N} (y_i(x^n) - t^n)(y_j(x^n) - t^n) \tag{3.38}$$

For a thorough derivation of why this is the optimal weighting, please see [1, p. 367].

Chapter 4

GAINN Software Overview

This chapter describes the GAINN software package. This first section will briefly discuss the software design. The rest of the chapter will take the reader through several screen shots and explain the features that are implemented. To be platform independent, the software was written in Java 1.4.2. This does cause the implementation to be slower than Matlab or a language that is not memory managed such as C++. However, for ease of deployment and modification, Java was the best option. Furthermore, since I intended the software to be readable by non-expert users, I did not want to use C++, since it is harder to understand.

4.1 GAINN Software Features

4.1.1 Dynamic GUI Model

One of my goals in the creation the GAINN software package was to make a user friendly, extensible framework for others. Thus, I was careful to architect the software in a manner such that adding new features would be a simple. Designing the GUI, in particular, was a difficult task since I wanted the program to be easy to modify. Often, GUIs involve a great deal of hard-coding and custom implementation that is not extensible and must be redone to make small changes. In GAINN, I created a framework that does not require any modifications to the GUI source code in order to

add new features. The **Constructor** interface is implemented by any property that may be selected through the GUI as shown in figure 4-1. An object exists for each parameter the end user may specify in GAINN. For example, the user has the choice of how to adjust the learning rate throughout training of the neural network:

1. Constant
2. Accelerate as $K * weight$
3. Converge as $\frac{1}{log(n)}$
4. Converge as $\frac{1}{K*n}$

There exists an object **LearningRateGenerator**, which implements **Constructor** and contains all methods necessary to determine the learning rate. The class **LearningRateGenerator** itself is abstract—as is every class in figure 4-1—and it contains an inner class for each of the different types of learning rate generators available. Typically, to create a new learning rate generator, I would have to create the new class, create a new entry in the GUI's drop down list, and then create additional if-then clauses to handle the selection of this new type of learning rate generator. Then, I would have to create a GUI screen to specify all of the parameters for this new learning rate generator. However, in the GAINN architecture, I simply create a subclass of LearningRateGenerator, and everything else will be handled by the current implementation. When the GUI is created for GAINN, each Constructor implementation is queried for the types of objects it contains. When LearningRateGenerator is queried, it returns the 4 methods mentioned above.

Each type of object may itself contain other parameters that need to be defined, and this is queried by the GUI constructor as well. The Constructor implementation contains methods **getValues** and **setValues**, which retrieve and set the information necessary to make an object that implements the Constructor interface. For example, when using the constant learning rate generator, a call to **getValues** on the constant learning rate object will return "rate" and "momentum". The actual value of these variables are set by the end user in the GUI, and a call to setValues is made when the

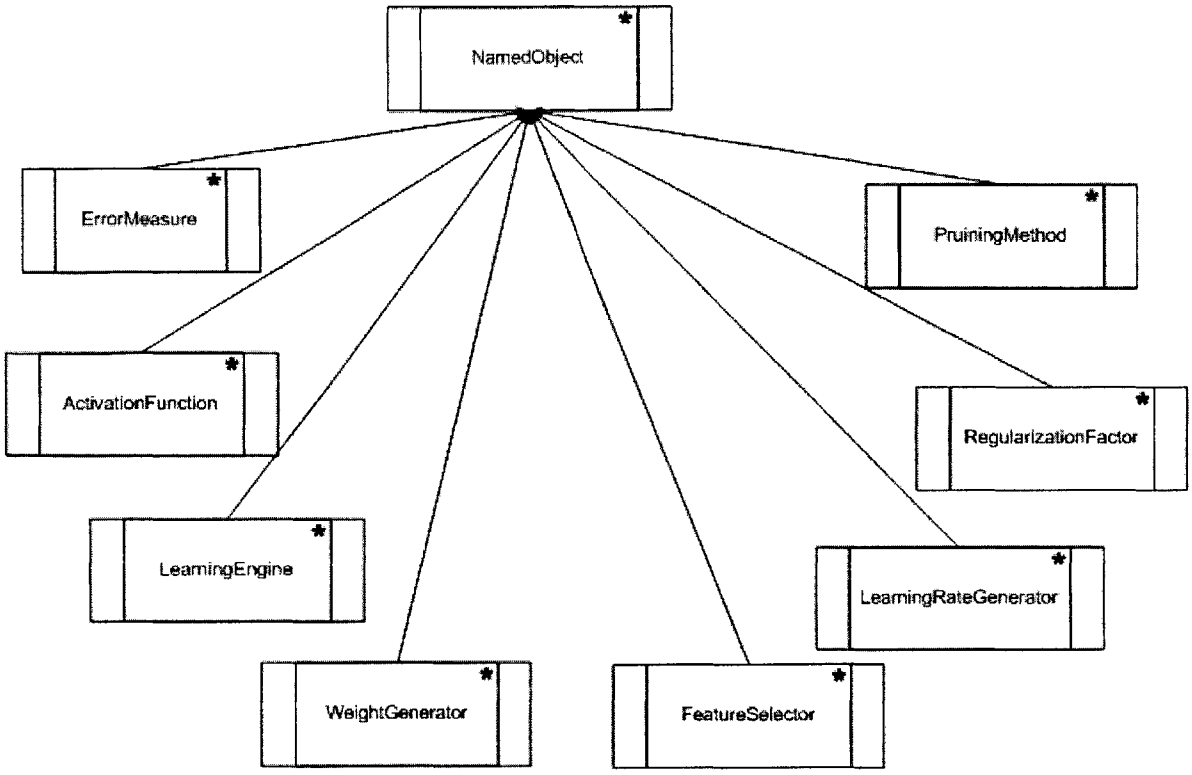

Figure 4-1: Classes that implement the NamedObject.

object is generated. All properties of a neural network that are set by the user in the GUI follow this pattern, so that the GUI can dynamically generate input boxes for each parameter that the end user must specify. This model allows GAINN to incorporate a new learning rate generator algorithm into software by creating one subclass in **LearningRateGenerator**. This implementation of the GUI was designed specifically to allow future additions to the GAINN framework without extensive changes to the original code base.

4.2 Data Loading

GAINN will read any dataset that is arranged in columns or rows and create a data matrix to be used in training and testing. The user can specify which features or items are to be ignored and what is to be used as the output. For efficiency, datasets are stored as **double** arrays; thus, it is necessary that all data in a dataset can be

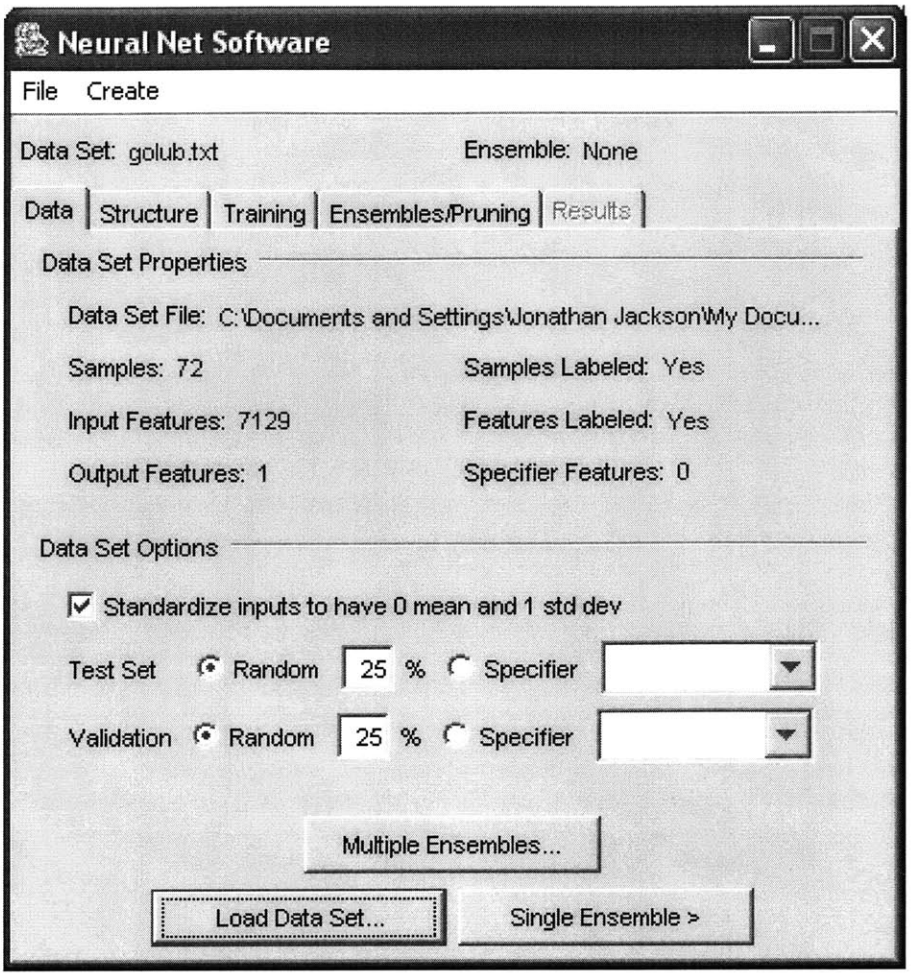

Figure 4-2: Loading a dataset.

parsed as doubles. GAINN is capable of training a dataset for any number of outputs or inputs, the only limiting factor being the memory and time available. For the purposes of 2-class separation (such as the datasets I explore in Chapter 5), it is assumed that all input data with a 0 output is of one class type, and everything else if of the second class type. It is necessary to make this distinction for binary outputs when creating ROC (receiver-operator characteristic) curves, which will be addressed in section 4.6.

The user specifies features that should be ignored. In the Golub dataset, for example, one of the rows contains a 1 if the patients had AML and another row contains a 1 if the patient had ALL. AML and ALL are mutually exclusive and

collectively exhaustive, so each case is either AML or ALL but not both. Therefore, if the user selects just one of these rows to be the output and does not ignore the other row, then there will be a feature that is perfectly correlated to the target value. This would make it trivial to classify the dataset using any technique.

There is also an option to standardize the inputs via a linear transformation such that the set of neurons to a particular input will have 0 mean and a standard deviation of 1. This transformation is very useful for starting the neural network in a position where the neurons are not saturated. Saturation can occur when the inputs to the activation function are in the range where the slope is near zero. If this occurs, changes to the weight settings may yield almost no change in the neuron's output, and it can be difficult for a training algorithm to overcome such a state.

4.2.1 Data splitting

Once a dataset is input, it is necessary to divide the set into the training set, validation set, and test set. The split may be determined by a percentage of the total data available or by using a feature in the data set as an indictor. The latter method is useful when using a dataset in which another researcher has marked which items were used for training and which ones were used for testing[1].

When choosing the training set and validation set sizes, it is important to keep the test set (the remainder of the dataset after determining the training set and validation set) of a large enough size such that it will be representative of its performance on future data. For example, if the test set used only contains one training item, we cannot conclude anything significant about the network's performance based on its classification of that one data item. I will show that the network's classification performance on the gene expression data can be greatly affected by how the dataset is split, because there is such a small number of samples.

[1]Such is the case with the dataset used by both Singh and Golub.

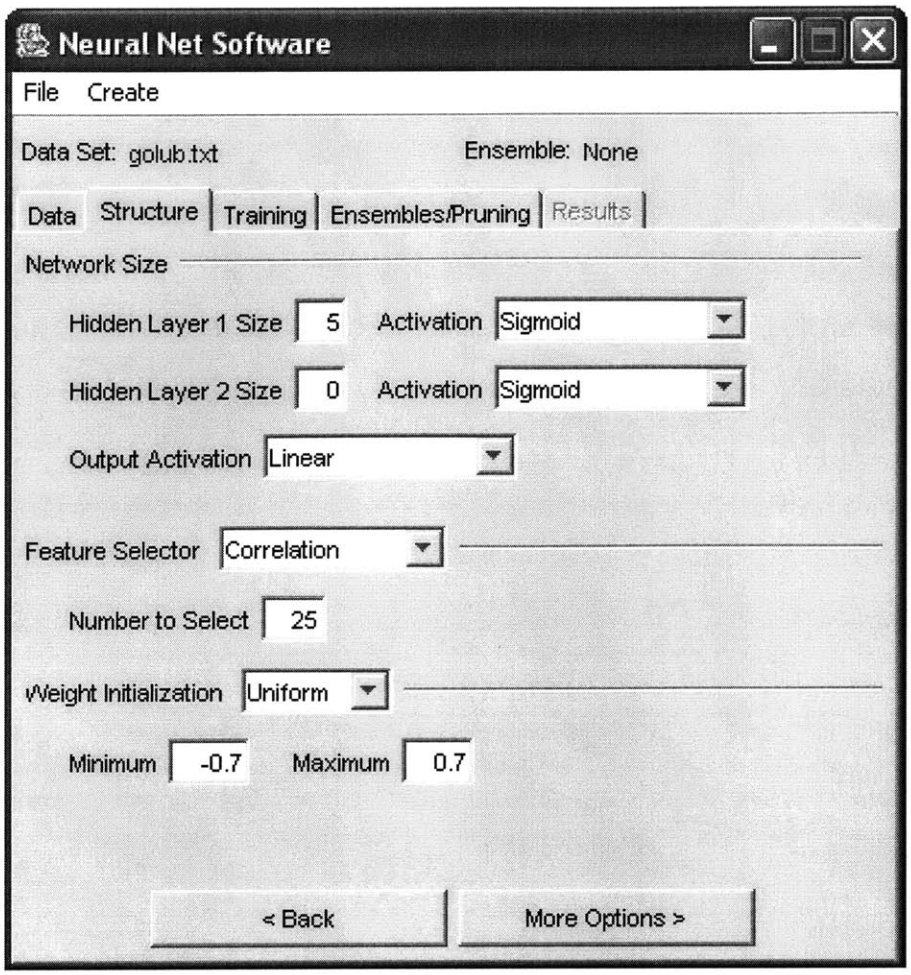

Figure 4-3: Choosing the network structure.

4.3 Network Structure

The next step in creating a neural network is to choose the structure of the neural network that we wish to train. Structure includes the number of neurons, the features, and the activation functions.

4.3.1 Hidden Neurons

The ability of feed-forward networks to classify non-linear datasets lies in the hidden neuron layers. In its current state, GAINN allows the user to specify two layers of hidden neurons as shown at the top of figure 4-3. More layers can easily be added to the GAINN code. However, based on the results of the gene expression classification,

gene expression analysis does not warrant more than two layers. In most cases, one layer is typically sufficient for excellent performance.

The user can specify how many hidden neurons to include in each layer[2]. There is no known formula to determine how many hidden neurons should be used for a given dataset. Theoretically, the number of hidden neurons necessary to approximate an arbitrary decision boundary given the number of inputs cannot be determined as a function of the number of inputs. For instance, when using threshold activation functions for the neurons, 2 hidden layers are necessary to approximate any decision boundary; the first hidden layer has d(2d+1) neurons and the second hidden layer has 2d+1 neurons [1]. There is, however, no known equation to calculate the weights and biases for such a network without knowing exactly what the decision boundary should be. That is, there is no training algorithm known that is guaranteed to find the correct decision boundary even if we know that the network is capable of creating such a boundary. When using sigmoid activation units, only one hidden layer is required to create an arbitrary decision boundary [15]. However, for gene expression classification, the boundaries are typically not so complex as to warrant a large number of hidden neurons. Empirically, I have found that there does not seem to be a significant association between the number of neurons, or even layers, and the classification performance of the neural networks.

4.3.2 Activation Functions

There are several activation functions for the neurons that may be used in GAINN: sigmoid, tanh, and linear. There are three other functions available but they do not have continuous derivatives and thus cannot be used in back-propagation[3],. Furthermore, the latter three functions yield less flexible networks. These functions only exist to test network outputs if the weights are loaded from a file.

The flexibility or capability of the neural network will depend on the activation function chosen. The sigmoid and tanh functions are equivalent in that a network

[2]To have only one hidden layer, a 0 can be specified for the number of neurons in layer 2.
[3]They are clamped functions, threshold, and sign function.

with the same number of neurons would be able to produce a network with the same decision boundary (if a linear transformation of the dataset was performed) [1]. Depending on the type of dataset, it may be necessary to use linear activation functions as the output neuron's activation function. When using sigmoid activation units, the output of a neuron is constrained to the range $[-1, +1]$. This is not a problem for the gene expression datasets because the outputs are binary values representing which class they belong to. When using other datasets, however, that are more typical regression sets (data generated by some function plus Gaussian noise), it is quite likely that in some dataset the output will be outside the range $[-1, +1]$. If this is the case, then the neural network will not be able to train properly using sigmoid or tanh activations for the output neurons.

4.3.3 Feature Selection

One of the key problems to solve with any machine learning technique is determining which features to use. For example, we could use all 6000+ gene levels from the Golub dataset, but a vast majority of the genes are expected to be uncorrelated to the classification task. In GAINN, we solve this problem by determining the correlation coefficient for each gene and the output. Then, GAINN will limit the number of features presented for training based on these values. The correlation coefficient is a simple metric relating the amount one variable appears to co-vary with another[4]. We expect that genes with the highest correlation coefficient would make the best feature set to use for classification, unless there is significant interaction among genes. I will demonstrate that using this method, GAINN can obtain excellent classification results.

With gene expression data, we typically only need a small number of genes (10-30) to classify the data well. There are other techniques that may be used to limit the number of features used in the dataset. One popular Bayesian technique is Automatic

[4]To calculate the coefficient, we calculate, $\frac{\sum xy - \frac{\sum y \sum x}{n}}{\sqrt{(\sum x^2 - \frac{(\sum x)^2}{n}) \times (\sum y^2 - \frac{(\sum y)^2}{n})}}$ where x is one of the features and y is the output.

Relevance Determination (ARD). This technique consists of using hyperparameters to assess the relevance of each input and limiting the weight adjustments based on these hyperparameters. This is a technique we plan to add to GAINN in the future. However, for the datasets tested here, it appears to be unneeded for classification tasks. One benefit of this Bayesian technique is that it does not require a limit on the feature set *a priori*. We can train a network with thousands of features and allow the hyperparameters to shrink the weights of irrelevant features. However, when using such a large number features, the training time for a neural network drastically increases.

4.4 Training Parameters

After setting the structure of the network, it is necessary to specify how to train this structure. There are several options available in GAINN, both in terms of the training algorithm and the parameters for that algorithm.

4.4.1 Update Technique

GAINN implements several training techniques that can be used:

1. Batch Back-Propagation
2. Online Back-Propagation
3. Listprop

These training techniques are based on the back-propagation algorithm described in section 3.5. Both online and batch training are available; the only difference between the two algorithms is when the weights are updated: after one training sample or one epoch, respectively. When using datasets that have a small number of samples, it is not likely that an improvement will be found from using online training as opposed to batch training. This is due to the fact that there will not be a high

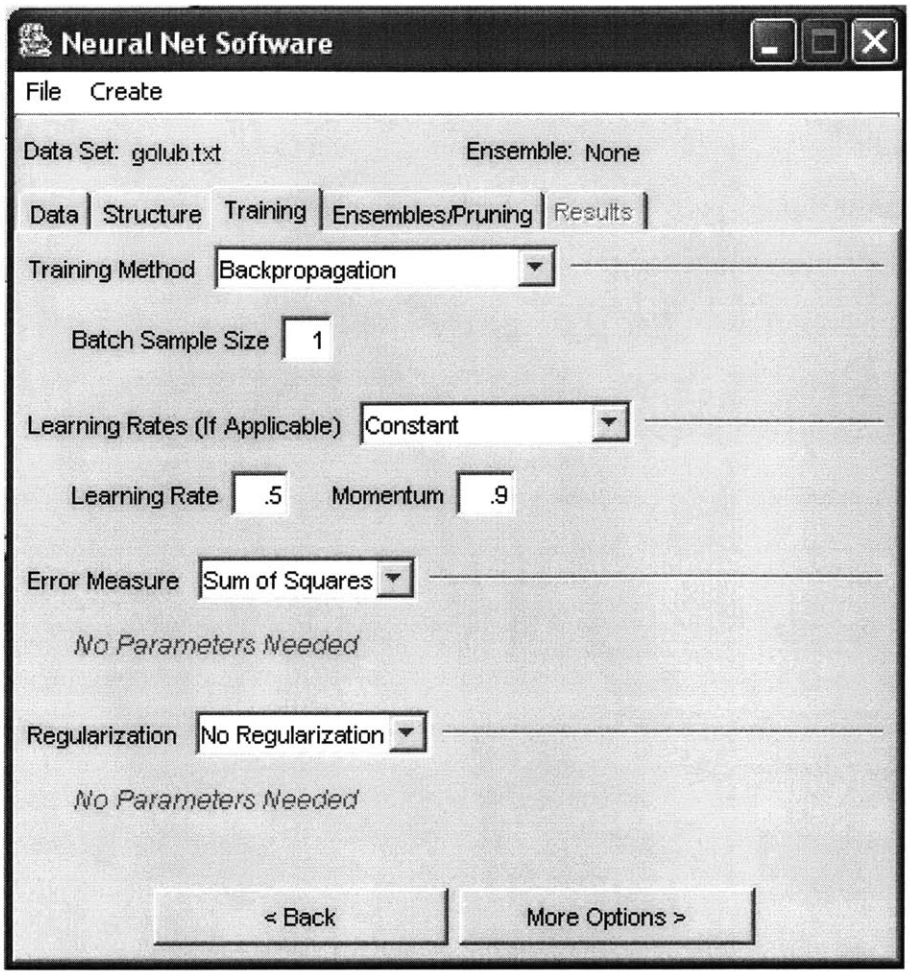

Figure 4-4: Choosing the network structure.

number of data points that are similar. Thus, we will need to examine every data point multiple times to properly train the neural network.

The other training algorithm available is called *Listprop*. Listprop works similar to online back-propagation training. However, if the network already outputs the correct value for a sample within a certain tolerance, then the weights are not updated for that sample [4]. This allows the network to avoid overtraining and possibly converge faster.

Stochastic training is also available, which does not go through the samples in order, but rather chooses a random order to present the sample or samples to the neural network. This technique is used to avoid getting stuck in a well on the error

surface, *i.e.* local minimum, that may be caused by updating weights for samples in the same order repeatedly.

4.4.2 Momentum and Learning Rate

As discussed in section 3.5, we need a way to determine the learning rate to use during training. GAINN implements 4 different learning rate modifiers:

1. Constant
2. Accelerate as $K * weight$
3. Converge as $\frac{1}{log(n)}$
4. Converge as $\frac{1}{K*n}$

The two convergence algorithms are based on the fact that improvements in the error function tend to be much larger at the beginning of training than later in the training progression. Because the neural network learns more as the training progresses, adjustments to the weights are not as helpful in improving the overall error. Therefore, larger steps down the error gradient may be taken at the beginning, when only the general direction is important. As the weight settings shift to decrease the error function, the slope of the error gradient will tend to decrease (going to zero if we reached the absolute minimum). Decreasing the learning rate as the training moves along will lower the chance that too large a step is taken and the error actually increases.

The acceleration algorithm is also focused on attempting to move down the error gradient as quickly as possible without moving too far and inadvertently increasing the error. The learning rate will be increased after each sample(s) as long as the overall error continues to decrease[5]. Thus, at the beginning of the training, the learning rate will continually increase allowing larger and larger steps to be taken. Whenever a weight adjustment is made and the error increases, the error rate is drastically

[5]The accelerating learning rate will examine the error after each weight update. If the error decreased, the learning rate is then increased. If the error increased, the learning rate is decreased.

reduced. As mentioned, the decrease in the error will lessen as the training progresses. This algorithm will take smaller steps further into the training since weight changes are more likely to increase the error than earlier in the training.

4.4.3 Error Measure

There are 4 error measures that are implemented in GAINN

1. Sum of Squares (Mikowski-2)
2. Absolute (Minkowski-1)
3. Cross Entropy
4. Minkowski-R

The optimal error measure to use for training the neural network depends on the dataset. When doing a typical regression analysis, Minkowski-R error or sum of squares is the best choice. As mentioned in section 3.7, the sum of squares error will find the conditional mean, whereas the Minkowski-1 error will find the conditional median. The benefit of the latter is that it is more resilient to outlying data

When training the neural network to classify into distinct classes, neither of the error measures just mentioned may be the optimal function to use. Rather, the cross-entropy error should perform the best for classification as described in section 3.7. This error function optimizes the network for class separation as opposed to a regression mapping.

4.4.4 Regularization

In some circumstances, it is desirable to regularize the neural network during training. There are two options available, weight decay and weight elimination. Regularization techniques are used in hope of keeping the neural networks more general. In some situations, generalization will lead to better classification results. Generalization techniques lower the weight changes by introducing penalty terms as discussed in section (3.6).

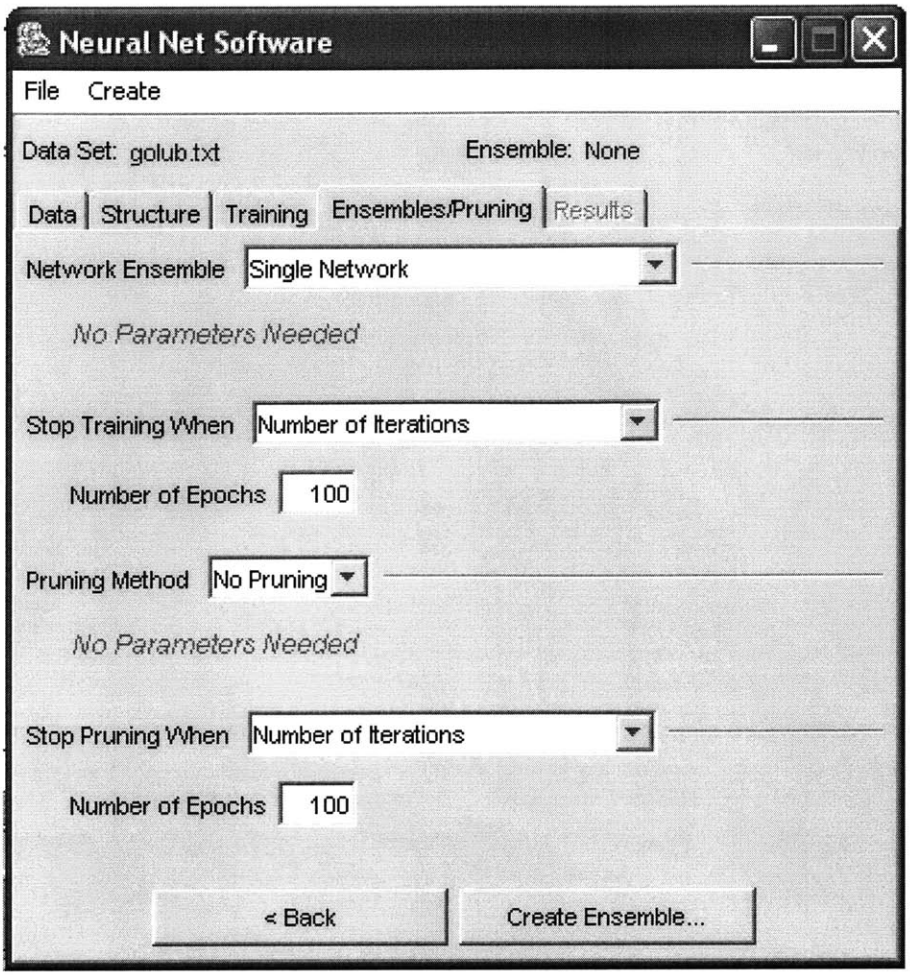

Figure 4-5: Ensembles and stopping criterion .

4.5 Training Endpoint

After the structure and algorithm have been set, we must determine when to stop the training algorithm. In addition, it may be desirable to create an ensemble of networks, which can lead to more accurate classification.

4.5.1 Stopping Criterion

Once the training algorithm and network structure have been specified, a stopping point for training must be determined. GAINN allows for several metrics:

1. Number of iterations

2. Minimum Training Error

3. Minimum Validation Error

4. Minimum Total Error

5. Percentage Decrease in Total Error

6. Percentage Decrease in Training Error

7. Percentage Decrease in Validation Error

The chosen error function in figure 4-5 not only affects the weight updates but also the metrics used to determine when to stop training. Minimizing the training error, when using a large number of iterations, can lead to bad generalization as it will tend to cause over-fitting. Minimizing the validation error is a better choice, as the samples in the validation set are not used for training the neural network. Therefore, the classification results of the validation set will be more indicative of how we expect the network to perform on the test set or other unseen samples. However, we may not always wish to use a validation set at all, so that we can train the network on more sample points; in this case, the stopping criterion can obviously not use the minimum of the validation error. Another metric I have implemented in GAINN is to use the total percentage decrease in one of the data splits. The intuition for this method is that the improvements in the error measure will decrease as the training progresses.

4.5.2 Ensembles

GAINN also allows the user to choose how many ensembles to create and how many to keep. The purpose of ensembles is to use each network to contribute to a final answer, thereby not discarding any networks that have been trained.

4.6 Neural Network Performance

GAINN provides the ability to examine the network performance after the training has completed. The results screen will show the user one of the most important

displays of discrimination when creating a two class discriminator: the ROC curve. The ROC curve, shows a tradeoff between specificity and sensitivity. Sensitivity is the probability that a patient with a disease will test positive for it, or, in the case of neural networks, be classified in the correct group:

$$\frac{TP}{TP + FN} \tag{4.1}$$

where TP is the number of true positives, and FN is the number of false negatives. Specificity is the probability that patients without a particular disease will test negative:

$$\frac{TN}{TN + FP} \tag{4.2}$$

where TN is the number of true negatives and FP is the number of false positives. We can obtain perfect sensitivity or specificity by blindly saying that all patients have a particular disease or all patients do not. However, we obviously hope to obtain much better classification than this. In the case of classifying between two different types of cancer, such as in the Golub dataset, we can simply denote one class as a positive test, and one class as a negative test. Then, each patient is classified into a group by examining the output of the neural network and using a particular cutoff value. For example, a cutoff value of 0.3 would mean that all samples that produced an output of 0.3 or less would be classified as "negative", while all samples that produced an output above .3 would be classified as "positive". GAINN will show the ROC curve for increments of 0.05 for the class separation. These 20 values will comprise the ROC curve, which plots the sensitivity versus 1 - specificity at different cutoff values.

The c-index is the area under the ROC curve and is a standard measure for determining how useful a medical test is; a higher c-index (1 being the maximum value) means a particular test is probably better than another[6]. We must define exactly what is meant by a positive test or a negative test. In the gene expression

[6]It is necessary to say "probably" here because, frequently, specificity and sensitivity are not valued the same for a particular test. For example, if we are only concerned with finding any and all patients with a particular disease, then we are more concerned with sensitivity. A diagnostic test that was more sensitive but had a lower specificity and lower c-index than another test may then be considered better.

datasets, the patients are all classified into two groups. One of the groups outputs has been set to 0, while the other has been set to not zero. In all datasets used here, the non-zero values are all equal to 1.

In addition, GAINN will display the error function's performance for all three sets, training, validation, and test. This is very useful for examining the behavior of the neural network throughout the training progression. Graphs are available for both the ROC curve and error function results.

4.7 Multiple Training Networks

The most convenient feature of GAINN is the ability to test all of the properties shown in figure 4-6 at one time. GAINN provides an easy-to-use interface (created dynamically) that allows the user to specify many options and range or values over which to train networks. GAINN will then train separate networks with each of the parameters specified. For example, if a user wanted to test the performance of neural networks using 3 different training algorithms and 2 different error functions, (s)he would simply specify these options in the dialogue shown in figure 4-6 and GAINN will produce six different networks and show the output. The training and test sets are kept the same through the creation of all the networks. This allows for extremely easy exploration of how different neural networks will perform and saves the end user a great deal of redundant effort. In addition, we often do not know what the best parameters to use with a particular dataset may be. When using GAINN, determining the optimal settings becomes much easier than using script-based programs, where the script may need to be edited over and over to test various parameters.

As shown in figure 4-7, once the training is completed, the results of all networks are shown in a single window, allowing the user to compare the performance of each network and see its characteristics. One can then select several networks to create an ensemble and explore the performance of the ensemble network as well.

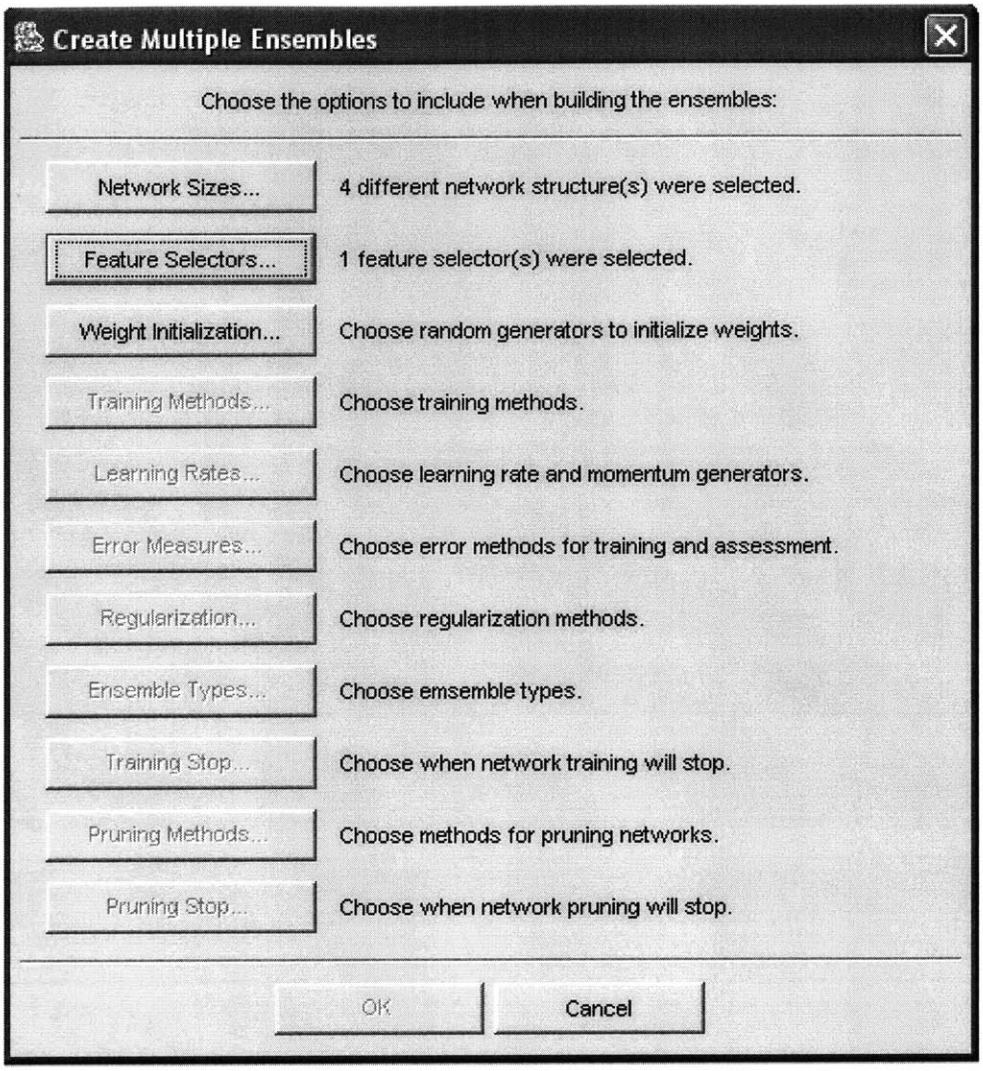

Figure 4-6: Creating multiple networks. Any parameter may be selected and numerous networks trained for that parameter.

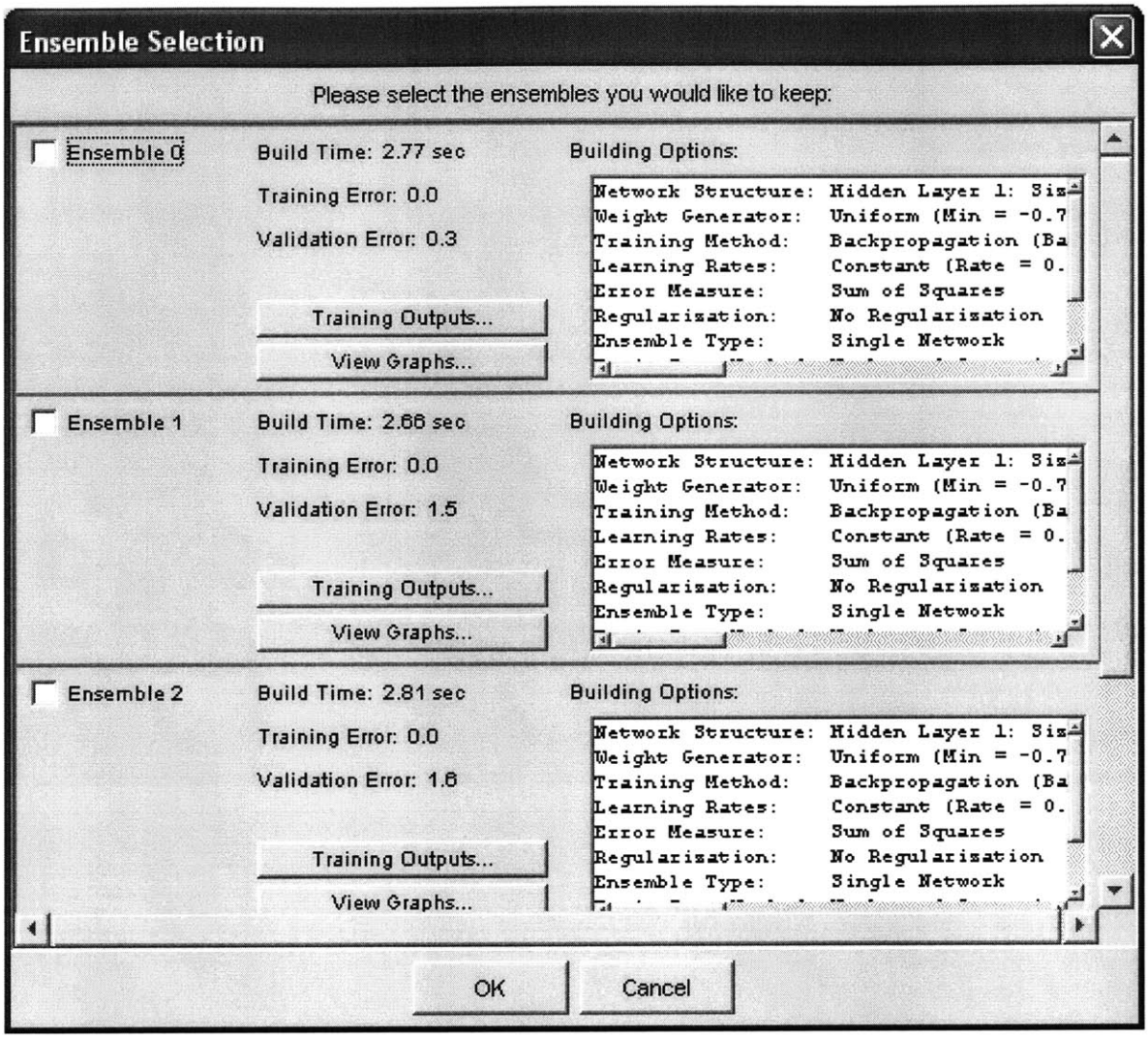

Figure 4-7: Viewing the results from multiple networks.

Chapter 5

Results

My testing methods were as follows: I tested GAINN's 2 class separation on two different datasets, one provided by Golub [9] and one provided by Singh [34]. These datasets contain gene expression data separating patients into two classes.

1. The Golub dataset contains samples of patients with either acute myeloid leukemia (AML) or acute lymphoblastic leukemia (ALL). The dataset contains 72 samples, each containing the expression level of 7129 genes. There are 47 patients with ALL and 25 with AML.

2. The Singh dataset contains patients that are either "normal" or diagnosed with prostate cancer. There were 102 samples: 52 from patients with prostate cancer and 50 from patients without prostate cancer. Gene expression levels were determined from 12,600 genes.

I will also report GAINN's performance on a multiple class data set provided by Ramsaway. This dataset contains gene expression data for patients with one of six different types of cancer. This dataset contains 64 samples and 16063 gene expression levels for 6 different types of cancer tumors [27].

5.1 Learning Algorithm

The following table shows the GAINN's performance on both the Golub and Singh datasets using the 4 different training algorithms. The network size specifies the number of features used[1]. For this test, the number of features used is also equal to the number of hidden neurons in the first layer. GAINN was able to accurately classify both datasets, but performed slightly better on the Golub dataset. In the original publishing, Golub was able to able to make a prediction on 29 out of 34 test samples with 100% accuracy on those 29 test samples. The remaining 5 samples were deemed unclassifiable. In the original publishing Singh was able to classify 90% of the samples correctly [34].

Learning Algorithm Performance			
Golub			
	Network Size		
	5	10	15
Online	0.9331	0.9386	0.9456
Batch	0.9574	0.9732	0.9804
Stochastic Online	0.9381	0.9410	0.9520
ListProp	0.9421	0.9395	0.9446
Singh			
Online	0.9186	0.9095	0.9161
Batch	0.9407	0.9387	0.9372
Stochastic Online	0.9142	0.9242	0.9372
ListProp	0.9241	0.9109	0.9314

Table 5.1: ROC area for various network learning algorithms on the test set.

I split the datasets into half training samples and half test cases. The values shown in table 5.1 are the average c-indexes over 25 trials. The results for any one cell varied widely during the trials as the largest standard deviation was 0.052 (found using online learning with a size of 10). This suggests either a large sensitivity to the training split, or a large sensitivity to the random starting weights. Because the ROC curve was perfect for the training data in 84% of the trials, the starting weights were most probably not the reason for the high deviation. Rather, the high

[1]Unless otherwise stated, the feature selection of the inputs was used by taking the genes with the highest correlation coefficient.

sensitivity to the data split comes from the small number of data samples. In the Golub dataset, some of the data were collected from a source that used a different technique to prepare the samples [9], and these samples were found harder to classify in the original publishing. If the data split contains more of these samples in the training set, then the performance of GAINN is better. If an unfavorable data split is randomly created, then GAINN will not be able to learn these patterns and the prediction will be worse.

Batch training outperformed online training in the Golub and Singh datasets. Each dataset was trained for 100 epochs. Since the data was collected by averaging 10 networks for each of the parameter settings, some of the 95% confidence intervals overlapped due to the high standard deviation. Therefore, I performed another experiment comparing online training to batch training for a 10-input neural network with 10 hidden neurons and sigmoid activation functions. Creating 50 networks for both online and training algorithms, I found that the batch online algorithm did outperform the online method as shown in table 5.2:

Golub			
	Mean C-index	Standard Deviation	95% Confidence Interval
Online	.931	.041	.919 - .942
Batch	.971	.023	.965 - .977
Singh			
Online	.905	.051	.891 - .919
Batch	.941	.034	.931 - .950

Table 5.2: Comparison of c-index for batch vs. online training in a 10 input neural network.

In both tests here, the batch learning algorithm outperformed the online training algorithm, and the results were statistically significant[2]. Online training updates the weights after each sample, so there were actually 7200 updates during training for the Golub datasets over the 100 epochs. On the other hand, the batch algorithm only updated after each epoch. Both of the training algorithms had a c-index of 1 on the

[2]Henceforth, differences in c-index averages will be deemed statistically significant if a t-test on the different means has a p-value of less than .05. That is, the null hypothesis will be that there is no difference in the means.

training set. However, this does not mean that the error was 0. When training for 100 epochs, the online neural networks were overfitting the dataset and not generalizing as well as the batch method.

5.2 Error Functions

I tested GAINN's performance on the Golub and Singh datasets with respect to the error functions used during training. The dataset was split into half training and half test data. The neural network had 10 features and 10 hidden neurons with sigmoid outputs. There was one output neuron that had either a sigmoid activation function or linear activation function. The results shown in table 5.3 are the average c-indexes over 20 trials.

Error Function Performance		
Golub		
	Activation Function	
	Sigmoid	Linear
Sum of Squares	0.9574	0.9180
Cross Entropy	0.9751	NA
Minkowski-1	0.9536	0.8592
Singh		
Sum of Squares	0.9387	0.9012
Cross Entropy	0.9528	NA
Minkowski-1	0.9363	0.8935

Table 5.3: ROC area for various error functions.

Once again, GAINN was able to obtain excellent classification results for both the Golub and Singh dataset with the Golub dataset performing slightly better. As mentioned in section 3.7, the cross-entropy error function is theoretically supposed to perform better on 2-class separation tasks than either of the other two error functions. This was true in both the Golub and Singh dataset when using sigmoid outputs. Because the cross-entropy error function uses a logarithm, this method is not capable of training on linear activation functions if the output is allowed to be negative. Therefore, it was not used to train the neural networks when linear outputs were used.

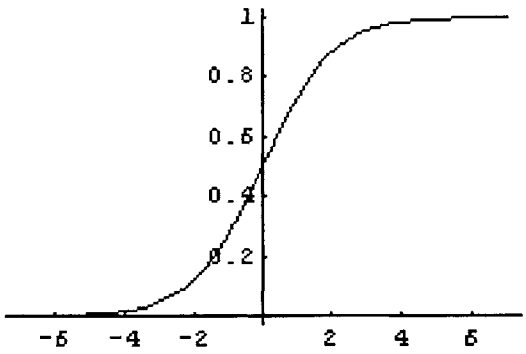

Figure 5-1: The sigmoid function.

Sigmoid neurons are considered "saturated" when the inputs reach either extreme of the function around +/- 3.5. At this point, the output of the neuron will be very close to 0 or 1, as shown in figure 5-1, and small changes in the weights could have little or no effect on the output of the neuron. The sigmoid function is better than the linear function for classification tasks because of this characteristic. If the outputs are not binary, however, sigmoid outputs might perform very poorly even if the output was restricted to the range [0,1].

Using a linear function as the output should perform worse than the sigmoid output, and this relationship is shown in table 5.3 It is interesting to note the poor performance of the Minkowski-1 when using linear outputs[3]. The Minkowski-1 error should be more resilient to over-fitting outlying data, because it finds the conditional median rather than the conditional mean. The sum-of-squares error—which is equal to Minkowski-2 error—will receive its highest contributions to weight changes from the data points with the largest error, because the error is squared. Minkowski-1 simply uses the absolute distance in one dimensional space. Based on the results, it appears that using the Minkowski-1 error may cause the neural network to underfit the dataset due to its lack of emphasis on outlying datapoints as compared to the sum-of-squares error measure.

[3]See section 3.7 for more information on the Minkowski error measure.

5.3 Training Speed

The neural networks used for gene expression classification in GAINN train extremely fast in real time—only a matter of seconds. This is due to the small number of inputs, hidden neurons, and small number of data samples. The real time speed of the training, however, conveys nothing about how the training error is progressing. The following tests were designed to gain intuition on GAINN's performance during training on these gene expression data sets.

Figure 5-2 depicts the root mean square (RMS) error on the training set through the training progression when using a constant learning rate on the Golub dataset. The learning rate of 1.0 took many more epochs to reach an RMS error below 0.04 as compare to learnrate rates of 1.5 and 2.0. However, the curve is much smoother than the ones for learning rates of 1.5 and 2.0. All of the learning rates above 1.0 converge to the same value of around 0.3. The learning rate of 2.0 provides a more erratic decent, as it sometimes moves too far down the gradient and actually increases the error rate. It appears as though the learning rate will not effect the network's performance on the training set, because the errors appear to be converging to the same value. In fact, only the network trained with a 0.1 learning rate had a different c-index on the test set in this case.

I also compared constant learning rate results to convergent and accelerated learning rates. However, there was no statistical significance in the difference of the test set RMS error or the c-index values when using the different learning rates modifiers. In the case of these datasets, GAINN can perform well regardless of which learning method it uses.

If datasets that were split unfavorably, however, there were some interesting differences. An unfavorable data split would typically lead to c-index of less 0.91 on the test set, while a favorable data split will lead to networks that score better than 0.95. In an unfavorable datasplit, both convergent learning rate modifiers would converge to an RMS higher than the constant learning rate or accelerating learning rate. This is due to the fact that the convergent learning rate decreases overtime and is not

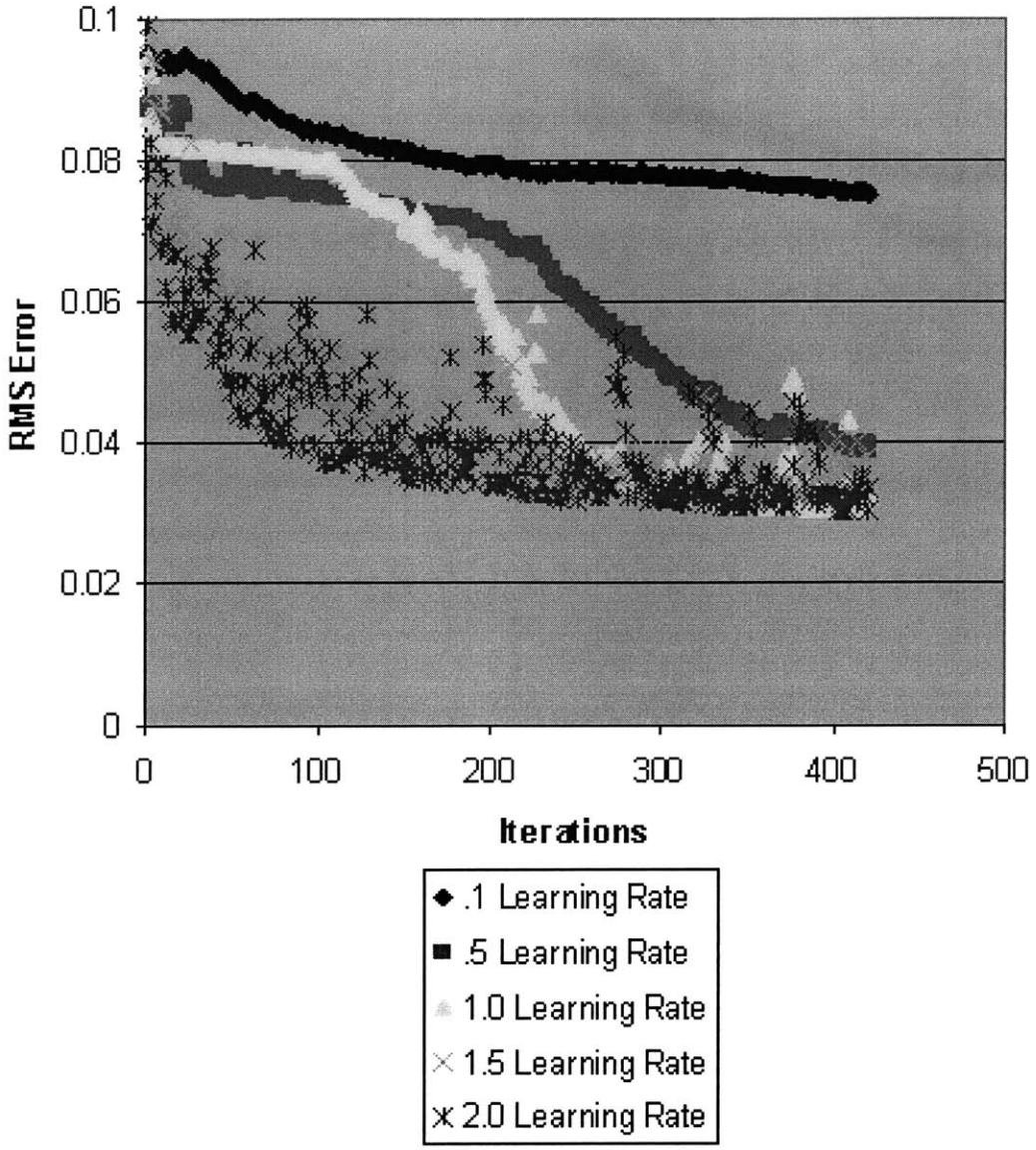

Figure 5-2: RMS error vs. training epoch iteration for different learning rates.

capable of learning the final difficult data points. Such an example of this type of training progression can be seen in figure 5-3. However, the constant and accelerating learning rates eventually did learn these difficult data points, and performed worse as a result.

In figure 5-3, we can see that around epoch 48, the network learns the final data point or points in the training set, and its error drops close to zero. The error for both the validation set and test set go up at this point, as the learning of this data

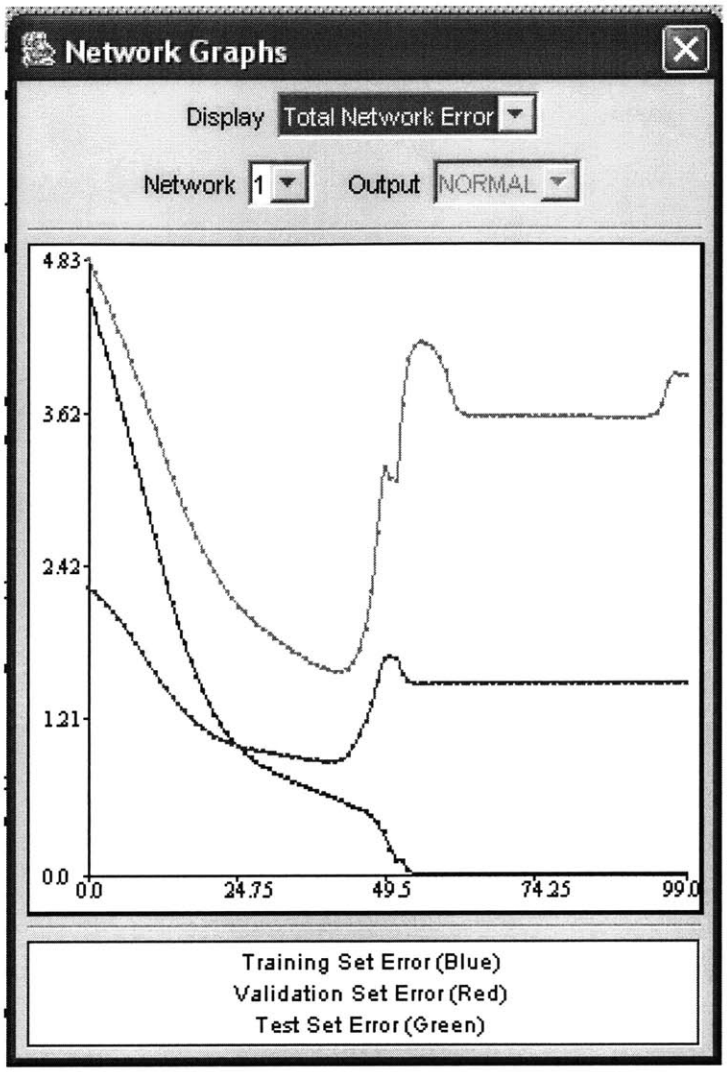

Figure 5-3: RMS error vs. training epoch iteration for a single training progression

point overfits the neural network to the training data.

5.4 Regularization

I tried using regularization during the neural network's training to see if there was a change in the classification performance[4]. Again, data were organized using half of the samples for training and half for the test results. The networks were trained using online training and a network size of 10. The coefficient of the regularization

[4]A discussion of regularization can be found in section (3.6).

term was 0.5, and the network were trained for 200 epochs.

Golub			
	Mean C-index	Standard Deviation	95% Confidence Interval
Online	.931	.041	.919 - .941
Regularized	.939	.047	.926 - .952
Singh			
Online	.905	.051	.890 - .919
Regularized	.919	.042	.907 - .930

Table 5.4: Comparison of c-index for online training using regularization.

Regularizing the neural networks did not yield statistically significant differences as compared to training the networks without regularization. The reason for this is that the sigmoid output neurons are typically operating in the non-linear regions after the first 50 epochs for almost all of the training samples, so updates made at this point are small regardless of regularization. This test was repeated with lambda values for regularization set to 0.25 and 0.75, but no significant changes were observed on the network performance.

5.5 Stopping Criterion

I also tested the network performance when different stopping criteria were used. However, once again, there did not appear to be a superior.

For these Golub test cases, the data were split into one third training, one third validation, and one third test set.

Golub			
	Training	Validation	Test
100 epochs	0.996	0.979	0.972
Minimum Training Error	1.000	0.962	0.958
Minimum Validation Error	0.981	0.978	0.965

Table 5.5: Comparison of c-index for different stopping criteria.

Contrary to the expectation that the minimum validation error should generalize better, training for 100 epochs performed the best. There was a higher standard

deviation in the results than when splitting the data into half training and half test samples. This does meet our expectation because the network has less data to train on and is therefore even more sensitive to how the data is split. With more complicated datasets, it is recommended that the validation error, or some other metric, be used to stop the training. Here, the network was able to reach a perfect ROC curve on the training set for all data splits. In addition, the network performed very well on the test set even when the training error was reduced to zero, which suggests that it is difficult to overtrain a neural network for classification of the Golub dataset.

5.6 Network Structure

As I have discussed, there is no known method for determining an optimal network structure given a particular dataset. It is not probable that such an equation could be derived based on the number of inputs either. The underlying function that maps the inputs to the outputs will dictate how complex the neural network structure should be. Here, I have created a surface plot of the c-index against the number of neurons in each of the two hidden layers. The left plot depicts networks trained with sigmoid activation functions, and 10 inputs. The right plot depicts networks trained with tanh activation functions and 20 inputs.

The data for the plots in figure 5-4 were collected using three separate data splits of 40 percent training data and 60 percent test data. The c-indexes were averaged for each configuration. The number of neurons in both layers ranged from 0 to 40 in increments of 5.

In both cases, the plot is highly irregular and does not suggest any one configuration or configurations tend to be optimal. Each point represents the average value over five trials.

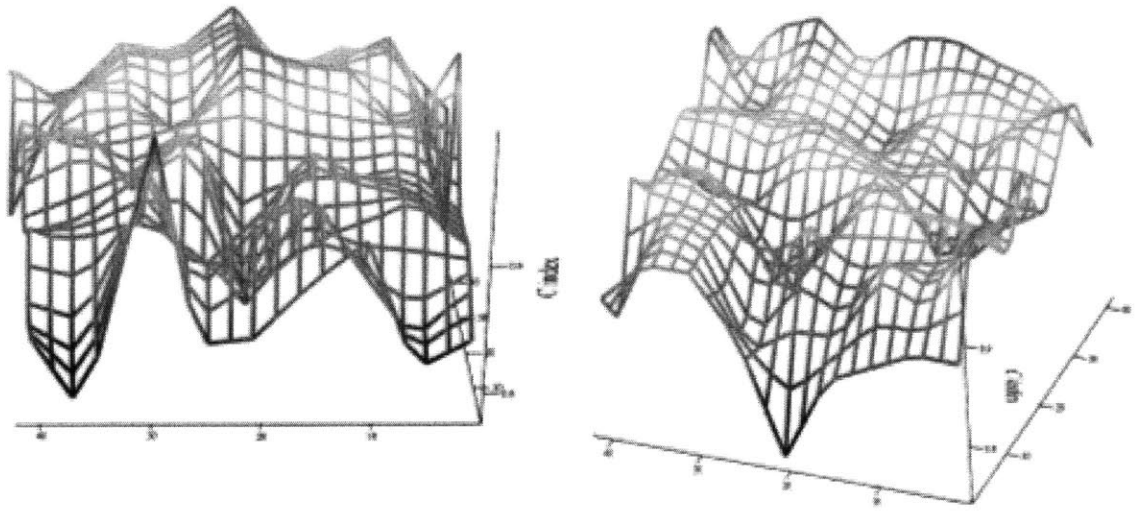

Figure 5-4: Surface plots of the c-index against the number of hidden neurons in each of two layers . The x-axis and y-axis range from 0 to 40.

5.7 Multiple Class Separation

While GAINN performed very well on the two-class separation problem, it did not perform as strongly on multiclass separation. For this test, I used 25 features, 20 hidden neurons, and 6 output neurons, all with sigmoid activation.

Ramsaway	
Tumor Type	C-Index
Breast	0.7660
Colorectal	1.0000
Lung	0.7730
Ovary	0.7628
Prostate	0.9152
Uterus	0.8984

Table 5.6: C-index values for multi class separation.

Here, the c-index is calculated one class at a time. This is not necessarily an appropriate measure, however, because a sample could be classified into two different classes simultaneously. In the 15 networks created, only 0.82 samples on average had two output neurons that were both above 0.5. When classifying the samples simply based on the highest output neuron's value, the median classification had an accuracy

of 100% on the training data and 81% on the test cases.

When using 50 features and 50 hidden neurons, the performance dramatically decreased as the network overfit the training data, and the classification accuracy dropped to 45%. In addition, the output's values were less conclusive. By less conclusive, I mean that in the networks that overfit the training data, there were more outputs in the range of [.2,.6] for one sample. A perfectly conclusive output for a sample would be one in which only one output neuron has a value of 1 and all others outputs are 0[5]. In general, misclassified datapoints are fare less conclusive. For example, in one misclassified test case all of the outputs were in the range of .25 to .34, yielding no useful information about that test case. In the better structured networks that generated the data for table 5.6, more of the outputs had only the correct neuron with an output above 0.1, and all other output neuron values were less than 0.1.

5.8 Feature Selection

I also tested the two-class datasets for performance while varying the numbers of features. These networks were trained with both batch and online training 5 times each and averaged. Similar to the optimal network structure search, no optimal number of features can be reported.

There are no general trends in the performance of the networks except that a very small number of features did not perform well. The standard deviation was large for these data points as well, though the standard deviation was slightly smaller in the range of 11 - 23 features. At some point, the neural networks should start to overfit the dataset given a huge number of inputs. However, even in a network with 100 inputs and 100 hidden neurons, the performance was not significantly different.

[5]However, this data point may still be misclassified.

Feature Selection Performance				
Number of Features	**Golub**		**Singh**	
	C-index	Standard Deviation	C-index	Standard Deviation
2	.878	.078	.891	.081
5	.937	.041	.915	.051
8	.956	.036	933	.049
11	.972	.035	.921	.042
14	.961	.032	.943	.039
17	.982	.021	.942	.031
20	.953	.027	.953	.036
23	.971	.038	.955	.041
26	.961	.045	.956	.056
29	.952	.044	.934	.036
32	.962	.031	942	.044
35	.951	.042	.949	.051

Table 5.7: ROC area for various numbers of features

Chapter 6

Contributions and Future Work

My intention in creating GAINN was to provide a robust neural network package that could be used for various classification tasks. I have implemented the neural network package and demonstrated the power of GAINN in classifying three gene expression microarray datasets.

While I discussed the many features that GAINN has available, there are many more than could be incorporated. A feature I have planned for the next release of GAINN is to implement the Bayesian automatic relevance determination learning algorithm. When using this algorithm, there is no need to limit the number of features before the training begins. It will be interesting to contrast the results of a network trained using ARD as compared to limiting the number of features based on correlation coefficient. In addition, I would like to build a genetic algorithm to combine parameters used to train the neural networks, not the neural network themselves.

Another goal was to create an open software package that researchers can use to classify many datasets (such a birth rate defects based on certain preconditions). GAINN requires little, if any, preprocessing steps and allows the end-user to easily vary many parameters of neural networks. GAINN graphically provides not only the error output, but also the ROC curve that is needed to evaluate performance on diagnostic tests. In addition, GAINN was designed as an easily extensible framework that future researches can easily add new features in the source code.

GAINN will remain an ongoing, open source project and the latest release and

source code will be available on the web. We hope that many others will be able to use GAINN as a tool for exploring neural networks as a means of classification.

Bibliography

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

[2] A.E. Bryson and Y.-C. Ho. *Applied Optimal Control.* Blaisdell, New York, 1969.

[3] Y.X. Chen, J.Y. Fang, H.Y. Zhu, R. Lu, Z.H. Cheng, and D.K. Qiu. Histone acetylation regulates p21(waf1) expression in human colon cancer cell lines. *World J Gastroenterol*, 10(18):2643–6, 2004.

[4] M.P. Craven. A faster learning neural network classifier using selective back propogation. In *Proceedings of the Fourth IEEE International Conference on Electronics, Circuits and Systems*, volume 1, pages 254–258, Cairo, Egypt, Dec 1997.

[5] G. Cybenko. Approximation by superspositions of sigmoidal functions. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1989.

[6] B. Farley and W.A. Clark. Simulation of self-organizing systems by digital computer. *IRE Transactions on Information Theory*, 4:76–84, 1954.

[7] K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Nerual Networks*, 2(3):183–192, 1989.

[8] D. Gershon. Microarray technology: an array of opportunities. *Nature*, 416:885–891, 2002.

[9] T.R. Golub, D.K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J.P. Mesirov, H. Coller, M.L. Loh, J.R. Downing, M.A. Caligiuri, C.D. Bloomfield, and E.S.

Lander. Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. *Science*, 286(5439):531–7, Oct 1999.

[10] S.R. Grey, S.S. Dlay, B.E. Leone, F. Cajone, and G.V. Sherbet. Prediction of nodal spread of breast cancer by using artificial neural network-based analyses of s100a4, nm23 and steroid receptor expression. *Clin Exp Metastasis*, 20(6):507–14, 2003.

[11] N. Haraguch, H. Inoue, K. Mimor, F. Tanaka, T. Utsunomiya, K. Yoshikawa, and M. Mori. Analysis of gastric cancer with cdna microarray. *Cancer Chemother Pharmacol*, Aug 2004. Epub ahead of print.

[12] D.O. Hebb. *The Organization of Behavior.* John Wiley & Sons, New York, 1949.

[13] J. A. Hertz, R. G. Palmer, and A. S. Krogh. *Introduction to the theory of neural computation.* Addison-Wesley, 1991.

[14] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[15] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[16] J. Khan, J.S. Wei, M. Ringner, L.H. Saal, M. Ladanyi, F. Westermann, F. Berthold, M. Schwab, C.R. Antonescu, C. Peterson, and P.S. Meltzer. Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks. *Nature Medicine*, 7(6):658–659, 2001.

[17] R. Kurzweil. The emergence of true machine intelligence in the twenty-first century (abstract). In *Proceedings of the 1993 ACM conference on Computer science*, page 507. ACM Press, 1993.

[18] Y. LeCun. A learning scheme for asymmetric threshold networks. In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

[19] E. Marder and V. Thirumalai. Cellular, synaptic and network effects of neuromodulation. *Neural Netw.*, 15(4):479–493, 2002.

[20] W.H. McCulloch, W.S.and Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–137, 1943.

[21] S.A. Minsky, M.L.and Papert. *Perceptrons.* MIT Press, Cambridge, 1969.

[22] R.M. Neal. *Bayesian Learning for Neural Networks.* Springer, 1996.

[23] R.M. Neal. *Neural Networks and Machine Learning*, chapter Assessing relevance determination methods using DELVE, pages 97–129. Springer-Verlag, 1998.

[24] M.C. O'Neill and L. Song. Neural network analysis of lymphoma microarray data: prognosis and diagnosis near-perfect. *BMC Bioinformatics*, 4(1):13, 2003.

[25] D.B. Parker. Learning logic. Technical Report TR-47, Center for Computation Research in Economics and Management Science, Massachusettes Institute of Technology, Cambridge, MA, 1985.

[26] M.P. Perrone. General averaging results for convex optimizations. In M.C. Mozer and et al., editors, *1993 Connectionist Models Summer School.* Lawrence Erlbaum Assoc Inc, 1994.

[27] S Ramsaway, KM Ross, ES Lander, and TR Golub. A molecular signature of metasis in primary solid tumors. *Nature Genetics*, 33(1):49–54, 2002.

[28] BD Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, New York, 1996.

[29] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[30] F. Rosenblatt. *Principals of Neurodynamics.* Spartan, New York, 1962.

[31] D.D. Rumelhart, G.E. Hinton, and R.J . Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[32] M. Schena, D. Shalon, and P.O. Davis, R.W .and Brown. Quantitative monitoring of gene expression patterns with a complementary dna microarray. *Science*, 270(5235):467–470, 1995.

[33] M. Schuster and K.K. Paliwal. Learning out time series with bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45:2673–2681, 1997.

[34] D. Singh, P.G. Febbo, K. Ross, D.G. Jackson, J. Manola, C. Ladd, P. Tamayo, A.A. Renshaw, A.V. D'Amico, J.P. Richie, E.S. Lander, M. Loda, P.W. Kantoff, T.R. Golub, and W.R. Sellers. Gene expression correlates of clinical prostate cancer behavior. *Cancer Cell*, 1(2):203–209, 2002.

[35] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[36] B. Widrow. Generalization and information in networks and adaline neurons. In M. Yovitz, G. Jacobi, and G. Goldstein, editors, *Self-organizing systems*, pages 435–461. Spartan, Washington DC, 1962.