8-2011

Implementation of a New Sigmoid Function in Backpropagation Neural Networks.

Jeffrey A. Bonnell
East Tennessee State University

Recommended Citation

Bonnell, Jeffrey A., "Implementation of a New Sigmoid Function in Backpropagation Neural Networks." (2011). *Electronic Theses and Dissertations.* Paper 1342. https://dc.etsu.edu/etd/1342

Implementation of a New Sigmoid Function in Backpropagation Neural Networks

---

A thesis

presented to

the faculty of the Department of Mathematics

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Mathematical Sciences

---

by

Jeff Bonnell

August 2011

Keywords: neural network, sigmoid, overtraining, binary classification

ABSTRACT

Implementation of a New Sigmoid Function in Backpropagation Neural Networks

by

Jeff Bonnell

This thesis presents the use of a new sigmoid activation function in backpropagation artificial neural networks (ANNs). ANNs using conventional activation functions may generalize poorly when trained on a set which includes quirky, mislabeled, unbalanced, or otherwise complicated data. This new activation function is an attempt to improve generalization and reduce overtraining on mislabeled or irrelevant data by restricting training when inputs to the hidden neurons are sufficiently small. This activation function includes a flattened, low-training region which grows or shrinks during backpropagation to ensure a desired proportion of inputs inside the low-training region. With a desired low-training proportion of 0, this activation function reduces to a standard sigmoidal curve. A network with the new activation function implemented in the hidden layer is trained on benchmark data sets and compared with the standard activation function in an attempt to improve area under the curve for the receiver operating characteristic in biological and other classification tasks.

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. Jeff Knisley, my thesis chair, for both the suggestion of this project and his continued support and advice during the arduous thesis process. Without his guidance, I would have found it impossible to make progress. I'd also like to thank each and every friend who has helped me during my years in the graduate program. I have been, in turns, stressed, sleepless, frantic, and lazy, and through it all they have been here to lend a hand when it was needed. I'd also like to thank the entirety of the department for their knowledge, openness, and helpfulness on questions both thesis-related and otherwise. Lastly, I'd like to thank my family for their love and support, without which this work could not have been done.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 BACKGROUND

Predictive modeling is a vital tool which finds applications in a wide range of scientific, mathematical, and financial fields. Biologists may employ predictive models and graph theory to classify RNA structures [8]. Police and forensic scientists take advantage of predictive models in fingerprint and face recognition software [2]. Meteorologists use predictive models to forecast weather, and market analysts use predictive models to forecast trends in the stock market [27].

In predictive modeling, a number of techniques are used to find the relevant information in available data and to construct a model for the prediction of currently unknown data. A variety of methods fall under the banner of predictive modeling. Linear regression uses the available data to construct a least-squares regression line as a model for predicting future data. Logistic regression can be used to predict the probability of a future event by fitting data to a logistic curve. Decision trees are another form of predictive model, using successive nodes to classify data as in a game of "20 questions" [15]. More complicated types of predictive models include support vector machines (SVMs) and artificial neural networks (ANNs). In a support vector machine, data is projected into a higher-dimensional space, where the data categories can be separated by a surface [6]. Future data is plotted into the same space, where it can then be categorized. This thesis focuses on artificial neural networks, which are highly non-linear forms of predictive models which are best suited to creating predictions by learning from complex relationships between available data [5].

1.1 Artificial Neural Networks

Artificial neural networks are suited to a variety of learning tasks, including classification, pattern recognition, and function approximation. We will be focusing solely on tasks requiring binary classification. To this end, we will be employing a feed-forward neural network called a multilayer perceptron (MLP)[9].

1.1.1 Network Inspiration and Structure

Artificial neural networks are a system of nodes called artificial neurons. Each artificial neuron in the network is a model inspired by the behavior of real neurons in the brain [9]. Natural neurons receive electrical signals through synapses, and when these signals surpass a threshold, the neuron will activate and fire a signal of its own [5]. Before considering a full neural network, we look at the behavior of a single artificial neuron.

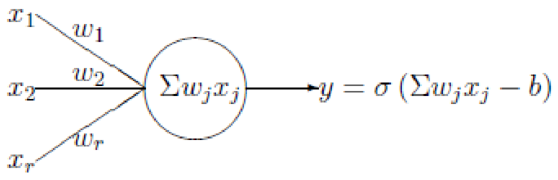

Figure 1: Artificial Neuron Diagram

An artificial neuron takes in a set of weighted inputs and applies an *activation function* to their sum. In Figure 1 above, x refers to an input, w to a weight, and b to a bias term. One of the most commonly used activation functions for artificial neurons is the logistic function

$$\sigma(x) = \frac{1}{1+e^{-\kappa x}}. \tag{1}$$

This function is a *sigmoid*, meaning that it is real-valued, differentiable, and strictly increasing. From this point forward, for ease of explanation, we will assume the parameter $\kappa = 0$. We now examine a graph of this activation function (Fig. 2).

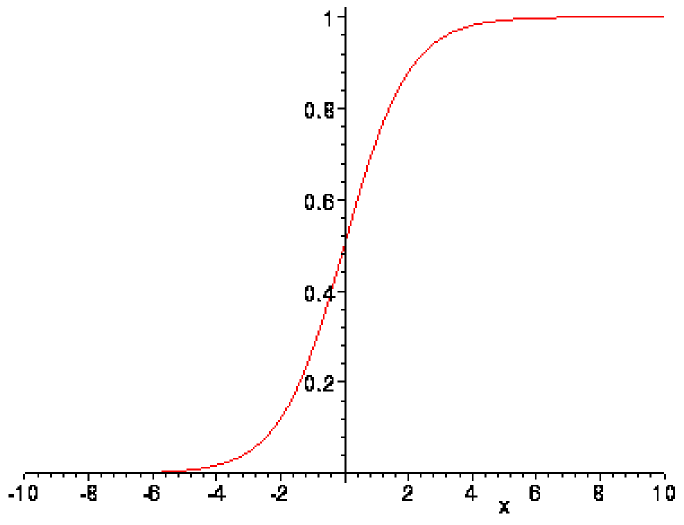

Figure 2: Standard Logistic Activation Function

A sigmoid activation function like the above ensures that the neuron can take in any real input and produce an output in the interval $(0, 1)$. In a biological sense, this could be interpreted as a probability function describing the chance that a neuron fires in a given time interval.

Now we can examine the structure of a three-layer artificial neural network. In the first layer, called the *input layer*, each neuron corresponds to one input to the network. Each node in the input layer is connected to each node in the second, or *hidden* layer, by a variable synaptic weight. Then, each node in the hidden layer is connected to each node in the third, or *output* layer by another variable synaptic weight. We will only be considering networks with one node in the output layer. A three-layer artificial neural network is displayed in Figure 3.

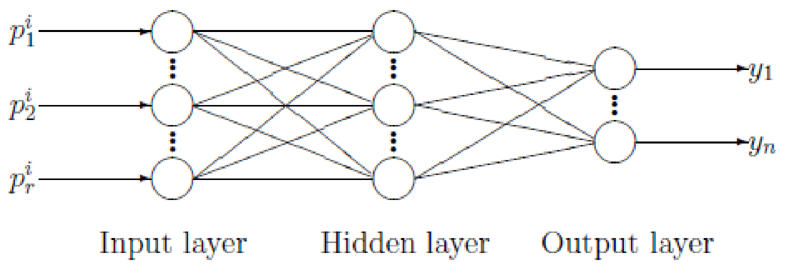

Figure 3: Three-Layer Artificial Neural Network

A neural network with m input nodes and 1 output node serves as a function with m inputs and 1 output. In the problem of binary classification, the goal is to use a set of m-dimensional training patterns with known outputs to train the network to partition this m-dimensional space such that each point is associated with either a positive or negative output. Once the space is partitioned, the network may be used to predict the classification of unknown patterns.

We now look at the prediction function defined by a three-layer feed-forward neural network using the standard logistic activation function. Given a pattern with m inputs and n nodes in the hidden layer, the input to the kth hidden node will be

$$\sum_{i=1}^{m} w_{ik}x_i + b, \tag{2}$$

where x_i is the ith input and w_{ik} is the weight between the ith input node and the kth hidden node. The output from the kth hidden node is given by

$$h_k = \sigma\left(\sum_{i=1}^{m} w_{ik}x_i + b\right), \tag{3}$$

where $\sigma(x)$ is the activation function described in equation 1.

Next, we calculate the input to the output node, which is given by

$$\sum_{k=1}^{n} \alpha_k h_k, \tag{4}$$

where h_k is the output from the kth hidden node and α_k is the weight from the kth hidden node to the output node. The output node's activation function is then applied to this value, and the output is given as

$$\begin{aligned} y &= \sigma\left(\sum_{k=1}^{n} \alpha_k h_k\right) \\ &= \sigma\left(\sum_{k=1}^{n} \alpha_k \sigma\left(\sum_{i=1}^{m} w_{ik}x_i + b\right)\right). \end{aligned} \tag{5}$$

1.1.2 Training via Backpropagation

We now examine the backpropagation learning algorithm [5, 9, 16, 26]. When presented with a set of m-dimensional input patterns $\mathbf{p}$, where some pattern p is given by $p = (x_1, x_2, x_3, \ldots, x_m)$, and the associated target output is t, we would like to minimize the error between the neural network output and the target value. The error is a function of the weights and is given by

$$E(\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_n}, \alpha) = \frac{1}{2}(y-t)^2, \tag{6}$$

where $\mathbf{w_k}$ is the m-dimensional vector of weights between the inputs and the kth hidden node, α is the vector of weights between the hidden nodes and the output, and y is the output of the network. Because we want to minimize this error function, we attempt to find weights which will give us

$$\frac{\partial E}{\partial \alpha_k} = 0 \tag{7}$$

and

$$\frac{\partial E}{\partial w_{ik}} = 0 \tag{8}$$

for all i and k. We will employ a pattern of gradient descent known as *backpropagation* to find the appropriate weights.

At each step of our iterative process, we must calculate the gradient of the error function E with respect to the weights by finding the partial derivative with respect to each weight. We first find the partial derivatives for the α weights.

$$\begin{aligned}\frac{\partial E}{\partial \alpha_k} &= \frac{\partial \left(\frac{1}{2}(y-t)^2\right)}{\partial \alpha_k} \\ &= (y-t)\frac{\partial y}{\partial \alpha_k} \\ &= (y-t)\frac{\partial \sigma\left(\sum_{k=1}^{n} \alpha_k h_k\right)}{\partial \alpha_k} \end{aligned} \tag{9}$$

To continue, we need the derivative of the function $\sigma(x)$. We have

$$\begin{aligned}\sigma(x) &= \frac{1}{1+e^{-x}} \\ \frac{d\sigma}{dx} &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1+e^{-x}-1}{(1+e^{-x})^2} \\ &= \sigma(x) - \frac{1}{(1+e^{-x})^2} \\ &= \sigma(x) - (\sigma(x))^2 \\ &= \sigma(x)(1-\sigma(x)). \qquad (10)\end{aligned}$$

Using this derivative, we continue with

$$\begin{aligned}\frac{\partial E}{\partial \alpha_k} &= (y-t)\frac{\partial \sigma\left(\sum_{k=1}^{n} \alpha_k h_k\right)}{\partial \alpha_k} \\ &= (y-t)(y)(1-y)h_k. \qquad (11)\end{aligned}$$

This value will be used in the weight update process at the end of the current iteration. Now we compute the gradient for the weights between input and hidden nodes as

$$\begin{aligned}
\frac{\partial E}{\partial w_{ik}} &= \frac{\partial \frac{1}{2}(y-t)^2}{\partial x_{ik}} \\
&= (y-t)\frac{\partial y}{\partial w_{ik}} \\
&= (y-t)\frac{\partial \sigma(\sum_{k=1}^{n} \alpha_k h_k)}{\partial w_{ik}} \\
&= (y-t)(y)(1-y)\frac{\partial (\sum_{k=1}^{n} \alpha_k h_k)}{\partial w_{ik}} \\
&= (y-t)(y)(1-y)\alpha_k \frac{\partial h_k}{\partial w_{ik}} \\
&= (y-t)(y)(1-y)\alpha_k \frac{\partial \sigma(\sum_{i=1}^{m} w_{ik}x_i + b)}{\partial w_{ik}} \\
&= (y-t)(y)(1-y)\alpha_k h_k(1-h_k)\frac{\partial (\sum_{i=1}^{m} w_{ik}x_i + b)}{\partial w_{ik}} \\
&= (y-t)(y)(1-y)\alpha_k h_k(1-h_k)x_i. \qquad (12)
\end{aligned}$$

In the process of stochastic backpropagation, after the gradient has been calculated using the first pattern in the training set, each weight in the network is updated [16]. The α weights are updated according to

$$\alpha_k \leftarrow \alpha_k - \eta\delta h_k, \qquad (13)$$

where η is a parameter called the *learning rate* and

$$\delta = (y-t)(y)(1-y)\alpha_k, \qquad (14)$$

as in equation (11). If a momentum parameter γ is included in the learning process, the weight update is increased by $\gamma\Delta(\alpha_k)_{j-1}$, where $\Delta(\alpha_k)_{j-1}$ is the size of the weight update from the previous iteration.

The w weights are also updated according to

$$w_{ik} \leftarrow w_{ik} - \eta\rho_k x_i, \qquad (15)$$

where again η is the learning rate and

$$\rho_k = h_k(1 - h_k)\alpha_k\delta, \tag{16}$$

with δ given in equation (14). Again, if a momentum factor γ is included in the learning process, the weight update is increased by $\gamma\Delta(w_{ik})_{j-1}$, with $\Delta(w_{ik})_{j-1}$ being the size of the weight update from the previous iteration.

One weight update is done successively for each pattern in the training set, and then the order of training patterns is randomized and the process continues to the next iteration. One pass through all of the training patterns is called a training *epoch*. Once training is complete, either due to some method of early stopping or the conclusion of a set number of training epochs, the final weight values are saved and can be used to calculate outputs for new patterns.

1.1.3 Difficulties with Artificial Neural Networks

ANNs have a number of strengths as a predictive modeling tool, chief among which is their ability to act as a universal approximator. Cybenko proved in 1988 that a neural network with two hidden layers can approximate any function to within any $\epsilon > 0$ [26]. This was followed by *Cybenko's Theorem*, which states that a feed-forward network with a single hidden layer using a sigmoid activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to any degree of accuracy $\epsilon > 0$ [4]. This result has since been extended for arbitrary, rather than strictly the standard sigmoid, activation functions [3].

Training an ANN, however, is not always a simple task. Through the process of gradient descent during backpropagation, it is possible for the network to converge on local minima of the error function that are not the desired global minimum. The use of simulated annealing, wherein the network is slightly perturbed as convergence slows, is one suggested remedy to this problem [25]. Another serious difficulty is the possibility of *over-training* on the set of training patterns [7, 11]. We would like to train the network to recognize the general patterns in the set, rather than overfitting every peculiarity of the data. The problem of overtraining can be serious when the training patterns contain a large amount of noise, irrelevant factors, mislabeled data, or other facets which do not correspond with the entire population of interest. In such cases, though training may be successful on the data given, the ANN will often fail to generalize well, causing poor performance on data outside the training set. It is for this reason that we propose a new activation function which may reduce the tendency to overtrain on problematic data sets.

1.2 Receiver Operating Characteristic

ANN performance in this thesis is evaluated using the area under the curve (AUC) of the receiver operating characteristic (ROC). This is a common metric for the performance of a predictive model under the task of binary classification, which is the case for all classification tasks in this thesis [10, 20].

The ROC is a plot of the classifier's *sensitivity* (true positive rate, given by $\frac{TP}{TP+FN}$, where TP = number of true positives and FN = number of false negatives) versus $(1 - \textit{specificity})$ (false positive rate, given by $\frac{FP}{FP+TN}$, where FP = number of false positives and TN = number of true negatives), for a particular classification threshold. The ROC is plotted for each possible threshold in the range $(0, 1)$, and the AUC is calculated by estimating the integral via the trapezoidal method.

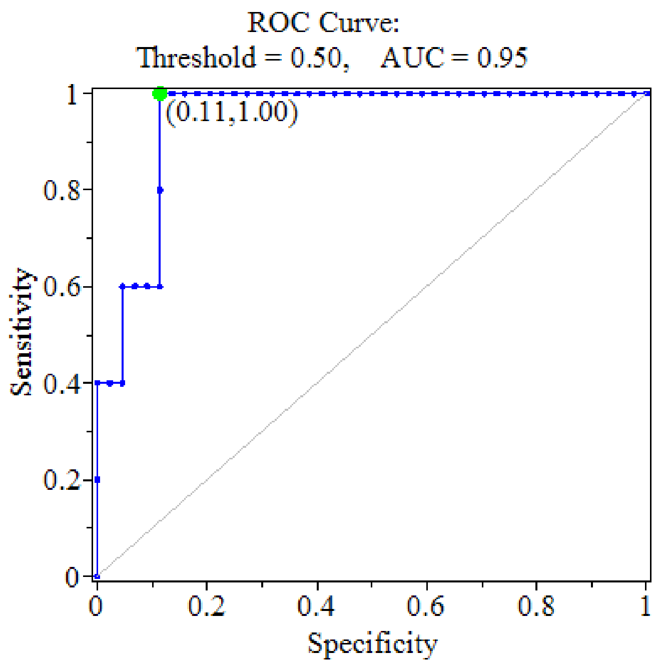

Figure 4: Example ROC Curve Including Optimal Threshold and AUC Value

A perfect classifier will display an AUC of 1.0 and purely random classifiers will display an AUC around 0.5. A higher AUC value indicates stronger classification performance. In Figure 4, the optimal threshold of 0.50 is given by the green dot. Note that it is the threshold closest to the upper left corner of the plot. It corresponds to a true positive rate of 1.00 and a false positive rate of 0.11.

2 METHODS

2.1 Implementation of New Activation Function

First, we have used the Python programming language to program a backpropagation artificial neural network with three layers as described above. The code may be found in the appendix. In most ways, this is a standard backpropagation neural network. It employs the method of gradient descent described above, including a term for momentum and a form of annealing. The annealing is accomplished by checking the change in the error term after a training epoch. If the change is below a specified value, each weight is perturbed by a small, normally distributed value centered around 0. For the classification done in this thesis, the amount of annealing has been kept small by universally setting the error threshold for annealing at 0.000002 and setting the standard deviation of the annealing term as $\frac{1}{5}$ of the learning rate. The primary difference between this network and a standard neural network is found in the activation function used for neurons in the hidden layer. Instead of the usual hyperbolic tangent or logistic activation functions, we employ the function

$$f(x) = \sigma(x+b) + \sigma(x-b) - 1, \tag{17}$$

where

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{18}$$

and b is a variable parameter. We notice that, when $b = 0$, we have

$$f(x) = 2\sigma(x) - 1, \tag{19}$$

which is just a rescaling of the standard logistic activation function from -1 to 1, rather than from 0 to 1.

Next, we examine the graphs of the activation function for three different values of the parameter b (Fig. 5).

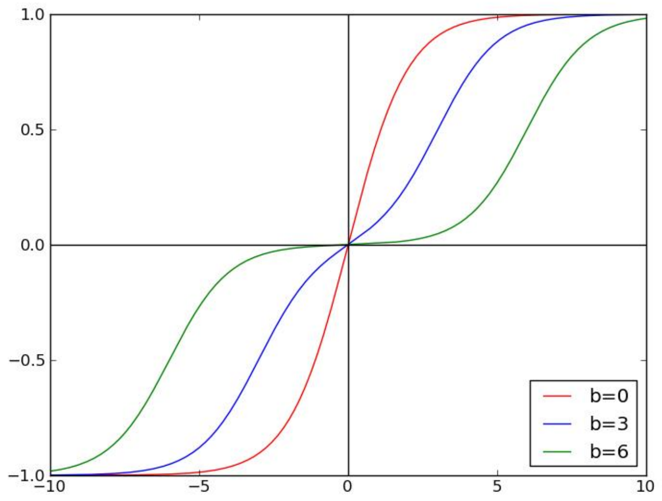

Figure 5: Hidden Layer Activation Function for Differing Values of b

We can see that, for $b = 0$, the activation function is just a rescaling of the standard logistic function. For larger values of the parameter, there is a “flattened” region of the curve near $x = 0$. The smaller derivative in this range creates what we hope to be a low-training region for small inputs to the hidden layer neurons.

The partial derivative of this activation function for the kth hidden neuron with respect to some input weight w_i must be computed in order to employ backpropagation training. First, let

$$a_k = \sum_{i=1}^{n} w_{ik} x_i \tag{20}$$

and

$$h_k = f(a_k). \tag{21}$$

We begin with the following:

$$\begin{aligned} h_k &= f(a_k) \\ &= \sigma(a_k + b) + \sigma(a_k - b) - 1. \end{aligned} \tag{22}$$

We take the partial derivative with respect to w_{ik} and get

$$\begin{aligned} \frac{\partial h_k}{\partial w_{ik}} &= \frac{\partial \sigma(a_k + b)}{\partial w_{ik}} + \frac{\partial \sigma(a_k - b)}{\partial w_{ik}} \\ &= \left[\sigma(a_k + b)(1 - \sigma(a_k + b)) + \sigma(a_k - b)(1 - \sigma(a_k - b))\right] x_i. \end{aligned} \tag{23}$$

Now, using equation (17) and substituting $1 - \sigma(a_k + b) = \sigma(a_k - b) - h_k$ and $1 - \sigma(a_k - b) = \sigma(a_k + b) - h_k$, we have

$$\begin{aligned} \frac{\partial h_k}{\partial w_{ik}} &= \left[\sigma(a_k + b)(\sigma(a_k - b) - h_k) + \sigma(a_k - b)(\sigma(a_k + b) - h_k)\right] x_i \\ &= \left[2\sigma(a_k + b)\sigma(a_k - b) - h_k(\sigma(a_k + b) + \sigma(a_k - b))\right] x_i. \end{aligned} \tag{24}$$

Finally, again referring to equation (17) and substituting $\sigma(a_k + b) + \sigma(a_k - b) = 1 + h_k$, we have

$$\frac{\partial h_k}{\partial w_{ik}} = \left[2\sigma(a_k + b)\sigma(a_k - b) - h_k(1 + h_k)\right] x_i. \tag{25}$$

2.2 First Method

In order to take advantage of the variable nature of this activation function, we have attempted two new methods of training. In the first, new steps are added to the standard backpropagation algorithm. The initial value of the parameter b is 0. During the forward stage of each iteration, the output of each hidden neuron is recorded for each training pattern. Then, the number of hidden layer outputs in a specified "lower-training" range is recorded. In this network, we have used the range $(-0.2, 0.2)$. Note that this range should be centered around 0, because this corresponds to the flatter, central portion of the activation function. The proportion of hidden layer outputs in the given range for this training epoch is recorded and compared against a desired range of proportions. If the current proportion is below this range, the value of the parameter b is increased by 0.05 in an attempt to catch a higher proportion in the low-training region. If the proportion is above the desired range and $b > 0$, the value of b is decreased by 0.05. This process is repeated at each iteration, and the final parameter value is saved along with the weights for use on test data.

2.3 Second Method

In order to perform the second method of training, we note that the parameter b is a variable and can therefore be included in the process of gradient descent used in backpropagation. First, we must calculate the partial derivative of the error function E with respect to b,

$$\begin{aligned}
E &= \frac{1}{2}\left(\sigma(\sum_{k=1}^{n}\alpha_k h_k) - t\right)^2 \\
\frac{\partial E}{\partial b} &= (y-t)(y)(1-y)\left(\sum_{k=1}^{n}\alpha_k \frac{\partial f}{\partial b}\right), \qquad (26)
\end{aligned}$$

where $\frac{\partial f}{\partial b}$ is the partial derivative of the hidden layer activation function, which we shall now calculate.

$$\begin{aligned}
f(x) &= \sigma(x+b) + \sigma(x-b) - 1 \\
\frac{\partial f}{\partial b} &= \sigma(x+b)(1-\sigma(x+b)) - \sigma(x-b)(1-\sigma(x-b)) \\
&= \sigma(x+b)(h_k + \sigma(x-b)) - \sigma(x-b)(h_k + \sigma(x+b)) \\
&= h_k(\sigma(x+b) - \sigma(x-b)). \qquad (27)
\end{aligned}$$

We return to our calculation of equation (26), substituting our derivative from (27), and get

$$\frac{\partial E}{\partial b} = (y-t)(y)(1-y)\left(\sum_{k=1}^{n}\alpha_k h_k\left(\sigma(\sum_{i=1}^{m} w_{ik}x_i + b) - \sigma(\sum_{i=1}^{m} w_{ik}x_i - b)\right)\right) \qquad (28)$$

If we perform standard gradient descent including this parameter, b quickly approaches 0, and the function reverts to the standard hidden layer activation. Because our goal is to avoid overtraining, we instead move *away* from the minimum in terms of b, adding the partial derivative instead of subtracting it. At each iteration, we update b according to

$$b \leftarrow b + \eta\frac{\partial E}{\partial b}. \qquad (29)$$

Backpropagation otherwise occurs in the standard way.

2.4 Comparison Between Networks

We have used three classification tasks to analyze the performance of our network. We will first discuss The Insurance Company (TIC) benchmark for data mining [24]. This data set contains 85 descriptive parameters for a large sample of customers, and our goal is to train our neural network to determine whether a given customer will have a mobile home insurance policy. This classification task is both slow and complicated, and we will compare our results with previous work to ensure that our network performs at least reasonably well on difficult classification tasks. Our second classification task involves artificial, generated data. We will be taking a classification task equivalent to the binary inclusive OR operator, adding inputs of pure noise, and intentionally mislabeling a small portion of the data [12]. The performance of our neural network incorporating the new hidden layer activation function will be compared with the performance of the same network with a standard activation function. Our final task involves the classification of biological data. Here, we will be classifying 3-phosphoglycerate kinase (3PGK) protein sequences from different phyla into their respective kingdoms of life [18, 19]. For this task, we will be comparing the performance of the new activation function against the performance of the standard function, and we will also be comparing our results to available benchmarking data.

In order to compare network performance, we will be performing multiple classification trials and finding the receiver operating characteristic (ROC) for each trial. We will then compare networks based on the area under the ROC curve (AUC).

3 RESULTS AND DISCUSSION

3.1 First Method Experimental Results

3.1.1 The Insurance Company Benchmark

This data set was used in the Computational Intelligence and Learning Cluster Challenge in 2000 [24]. The training set consists of a set of 5822 customer records, each containing 85 customer attributes, including sociodemographic data and product ownership statistics. The target variable was the binary classification of whether the customer had a mobile home insurance policy. The evaluation data set was a similarly formatted file containing data on the same attributes for a separate set of 4000 customers.

Due to the varied nature of the attributes, instead of working with the raw information, the data for each descriptive attribute were first normalized according to the formula

$$z = \frac{x - \bar{x}}{s_x}, \tag{30}$$

where $\bar{x}$ is the mean value for attribute x and s_x is the sample standard deviation of the data for attribute x.

Another problem with the data set is its unbalanced nature. Because only 348 of the 5822 customers in the training set have a mobile home insurance policy, the network may overtrain on the negative portion of the training set. Therefore, in each training run, all 348 members of the positive set are included in our training as well as a simple random sample of 348 members of the negative set.

Two training methods were attempted. In the first, for each training run, a simple random sample of 150 customer records was taken from the training set to be used as a validation set for early stopping of the training session. If the error of the validation set increased for 50 consecutive training epochs, the training was stopped and the weights of the network returned to their states corresponding to the epoch with the minimum validation set error. In this method, 170 neurons were placed in the hidden layer, the learning rate was set to 0.05, and the momentum factor was set to 0.1. Due to the high number of neurons in the hidden layer, the proportion of hidden layer outputs in the range $(-0.2, 0.2)$ never fell below the desired proportion of 0.1 for any training run. Therefore, the value of the parameter b in the hidden layer activation function never rose above 0, and we consider only the standard activation function.

The AUC for the ROC was calculated for the training and evaluation sets following each training run, and the results are summarized Table 1.

Table 1: AUC Results for Early Stopping on TIC Data

Number of Runs	Training AUC Mean	Eval. AUC Mean	Eval. AUC St. Dev.
100	0.813417	0.700114	0.013174

These AUC values compare reasonably well with past neural network classifications on this data, and we can continue to the next training method [28]. For this method, training was done in the same way as the previous method except for two key differences. First, 50 neurons were used in the hidden layer. Second, no validation set was used. Instead of early stopping, each training run consisted of 100 training epochs. Again, the large number of hidden neurons results in an unchanging hidden layer activation function, so we consider only the standard network.

The AUC for the ROC was calculated for the training and evaluation sets following each training session, and the results are summarized below in Table 2.

Table 2: AUC Results Following 100 Epochs on TIC Data

Number of Runs	Training AUC Mean	Eval. AUC Mean	Eval. AUC St. Dev.
100	0.990502	0.660868	0.017270

These results are again comparable with those found in [28], wherein some similar methods were used and a 95% CI of $(0.588, 0.740)$ was found for the AUC of the evaluation set ROC.

3.1.2 Modified Inclusive OR Operator

Now we look at training the neural network on a randomly generated data set that mimics a modified version of the binary inclusive OR function [12]. Each pattern in the training and evaluation data sets consists of four inputs and one output. Each of the four inputs is generated using a normal distribution centered at 0 with standard deviation 10 ($N(0, 10)$). The first two inputs are purely noise. If either of the remaining two inputs is positive, then the pattern will be classified in the positive set. If both are negative, then the pattern will be classified in the negative set. Then, as an additional obstacle to learning, one important change is made. In the training set, if exactly one of the third or fourth inputs for a given pattern is positive, the pattern will be misclassified as negative with probability $p = 0.10$. This is done only for the training set, and none of the evaluation set patterns are misclassified. The classification rules for the training set are summarized Table 3 below.

Table 3: Modified Binary Inclusive OR Classification (Training Set)

x_1	x_2	x_3	x_4	Classification
-	-	-	-	-
-	-	-	+	p(+)=.9,p(-)=.1
-	-	+	-	p(+)=.9,p(-)=.1
-	+	-	-	-
+	-	-	-	-
-	-	+	+	+
-	+	-	+	p(+)=.9,p(-)=.1
+	-	-	+	p(+)=.9,p(-)=.1
-	+	+	-	p(+)=.9,p(-)=.1
+	-	+	-	p(+)=.9,p(-)=.1
+	+	-	-	-
-	+	+	+	+
+	-	+	+	+
+	+	-	+	p(+)=.9,p(-)=.1
+	+	+	-	p(+)=.9,p(-)=.1
+	+	+	+	+

Because half of the input variables are purely noise and an average of 5% of the training patterns are misclassified, this data provides an opportunity to compare the performance of a network using the new activation function with the performance of a standard network.

For each training run, a new set of 100 training and 100 evaluation patterns was generated following the above rules. Each network was trained on the given training set for 1000 epochs using a learning rate of 0.3 and a momentum factor of 0.1, with 4 neurons in the hidden layer. The desired range of proportions for hidden layer outputs between -0.2 and 0.2 was $(0.8, 0.12)$. 500 training runs were performed, and the results are summarized in Table 4. AUC results for the modified network are listed as AUC_1, and AUC results for the standard network are listed as AUC_2.

Table 4: AUC Results For Training on Modified Inclusive OR Data

	Mean	Standard Deviation
AUC_1	0.953136	0.040583
AUC_2	0.943934	0.040025
$AUC_1 - AUC_2$	0.009203	0.044836

If we make the assumption that the AUC difference results given by the neural networks are normally distributed, then we may perform a paired t-test for difference in mean AUC between the two networks [13]. The test will be performed in SAS version 9.2 using a 95% confidence level. The results are shown below in Table 5.

Table 5: Paired t-Test for Difference in Means on Binary OR Data

Mean Difference	St. Dev.	Sample Size	t Statistic	$P(t > t^*)$	95% CI for Mean Difference
0.009203	0.044836	500	$t = 4.5895$	$P < 0.0001$	$(0.005263, 0.013142)$

If we make the assumption of normality, we can conclude that there is a statistically significant difference between the AUC results for a network using the new activation function and one using the standard activation function, though the new network does not always outperform the old one. We now have reason to examine the change in performance on real data sets.

3.1.3 3-Phosphoglycerate Kinase Protein Sequences

We now turn to a problem of biological classification. We begin with 3-phosphoglycerate kinase (3PGK) protein sequences for members of different phyla, and we wish to classify these members either inside or outside a particular kingdom [18, 19]. We will focus on two phyla in particular: Proteobacteria, from the kingdom Bacteria, and Euglenozoa, from the kingdom Eukaryota. We will also be considering three distance measures to quantify the protein sequences: BLAST distance matrix [1], Smith-Waterman [21, 23], and Local Alignment kernel [22]. These combinations result in six training sets, one for each phylum/distance measure combination, and the corresponding six evaluation sets.

There are 70 members in each Proteobacteria training set, 61 members in each Proteobacteria evaluation set, 82 members in each Euglenozoa training set, and 49 members in each Euglenozoa evaluation set. The Proteobacteria training sets include 43 positively classified members of the kingdom and 27 negatively classified members outside the kingdom, and the Proteobacteria evaluation sets include 30 positive members and 31 negative members. The Euglenozoa training sets include 38 positively classified members of the kingdom and 44 negatively classified members outside the kingdom, and the Euglenozoa evaluation sets include 5 positive members and 44 negative members.

The data were compiled by taking each protein sequence in the training set and computing the distance measure to each member of the training set, creating a matrix of distance scores. The evaluation sets were created by finding the distance measure from each member of the evaluation set to each member of the training set.

For each training set, the networks were set up with 2 neurons in the hidden layer and trained for 500 epochs. A learning rate of 0.3 and a momentum factor of 0.1 were used, and an ideal proportion of "low-training" hidden layer outputs was set to the range $(0.08, 0.12)$. 200 training runs were done for each set, and the mean, standard deviation, and 95% CI for the evaluation set AUC were calculated. We also compared the AUC results to those listed as the benchmark value.

We will look first at the Proteobacteria data. The results after training for the BLAST distance matrix set are shown below (Table 6).

Table 6: AUC Results Following 200 Runs on BLAST Proteobacteria Data

	AUC Mean	AUC St. Dev.	95% CI for Mean AUC
New Activation Function	0.917091	0.002216	$(0.916782, 0.917400)$
Standard Activation Function	0.915258	0.002216	$(0.914949, 0.915567)$
Benchmark AUC [19]	0.9022		

Next, we look at the AUC results after training on the Smith-Waterman set (Table 7).

Table 7: AUC Results Following 200 Runs on SW Proteobacteria Data

	AUC Mean	AUC St. Dev.	95% CI for Mean AUC
New Activation Function	0.901118	0.001512	$(0.900907, 0.901329)$
Standard Activation Function	0.899199	0.001858	$(0.898940, 0.899458)$
Benchmark AUC [19]	0.8935		

Next, we examine the results for the Local Alignment kernel set (Table 8).

Table 8: AUC Results Following 200 Runs on LA Proteobacteria Data

	AUC Mean	AUC St. Dev.	95% CI for Mean AUC
New Activation Function	0.902855	0.001370	$(0.902664, 0.903046)$
Standard Activation Function	0.902715	0.001470	$(0.902510, 0.902920)$
Benchmark AUC [19]	0.8957		

The networks both outperform the benchmark AUC value for each distance measuring method. The network with the new activation function outperforms the old network by a small but statistically significant margin for the BLAST and Smith-Waterman sets, but there is not enough evidence to show a difference in performance on the Local Alignment kernel set.

We now examine the Euglenozoa data, first looking at the BLAST distance measure in Table 9.

Table 9: AUC Results Following 200 Runs on BLAST Euglenozoa Data

	AUC Mean	AUC St. Dev.	95% CI for Mean AUC
New Activation Function	0.767568	0.013110	$(0.765740, 0.769396)$
Standard Activation Function	0.738886	0.017606	$(0.736432, 0.741341)$
Benchmark AUC [19]	0.8318		

We move next to the Smith-Waterman distance measure, with results displayed in Table 10.

Table 10: AUC Results Following 200 Runs on SW Euglenozoa Data

	AUC Mean	AUC St. Dev.	95% CI for Mean AUC
New Activation Function	0.773114	0.008522	$(0.771925, 0.774302)$
Standard Activation Function	0.773682	0.009256	$(0.772391, 0.774972)$
Benchmark AUC [19]	0.8045		

Finally, we look at the data using the Local Alignment kernel in Table 11.

Table 11: AUC Results Following 200 Runs on LA Euglenozoa Data

	AUC Mean	AUC St. Dev.	95% CI for Mean AUC
New Activation Function	0.790773	0.011697	$(0.789142, 0.792404)$
Standard Activation Function	0.796023	0.011891	$(0.794365, 0.797681)$
Benchmark AUC [19]	0.8182		

The results for the Euglenozoa training sets are less encouraging than those for the Proteobacteria sets. The networks both underperformed the benchmark AUC values in every case. The new network outperformed the standard one on the BLAST data set, but the standard network showed better performance on the LA data set. There was not enough evidence to determine a difference in the performance of the two networks on the SW data.

3.2 Second Method Experimental Results

Using the second method, gradient ascent in terms of the parameter b, we investigate the 3PGK data sets. First, we examine a common graph of the error during backpropagation using this method (Fig. 6).

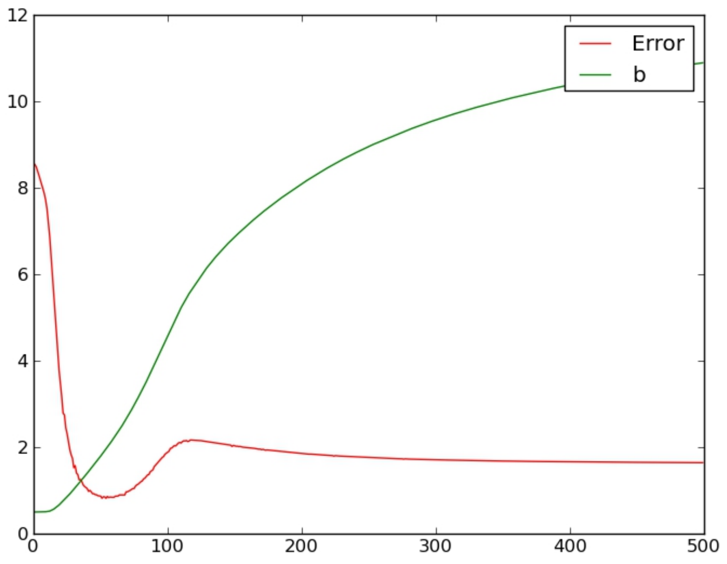

Figure 6: Backpropagation Error and b-Value vs. Iteration Number

We note that, as b increases, the error stops approaching the global minimum, begins to increase, and then falls into a local minimum larger than 0. The hope is that such training forces the network to learn the "important" patterns in the data while only undergoing small, perturbative training on the details.

We did 200 training runs of 500 epochs and 50 training runs of 5000 epochs on each 3PGK data set. A learning rate of 0.05 and a momentum of 0.1 were used, and only one neuron was included in the hidden layer. With additional hidden layer neurons, overtraining may persist despite the implementation of the new method. The initial value for b was set to 0.5. Our results for the Proteobacteria data are listed in Table 12 along with benchmark results for ANN, SVM, and logistic regression. The highest performing classification for each set is colored blue.

Table 12: Mean AUC Results Following for Proteobacteria Using 2nd Method. Benchmarks from [19]

	BLAST	SW	LA
95% CI, Method 2, 500 Iter.	(0.962368, 0.962740)	(0.953664, 0.954099)	(0.948620, 0.949101)
95% CI, Method 2, 5000 Iter.	(0.941544, 0.942006)	(0.945393, 0.946113)	(0.947197, 0.948717)
95% CI, Standard, 500 Iter.	(0.915242, 0.915887)	(0.899367, 0.899977)	(0.902098, 0.902547)
95% CI, Standard, 5000 Iter.	(0.916188, 0.917322)	(0.900121, 0.901492)	(0.902488, 0.903964)
Benchmark ANN	0.9022	0.8935	0.8957
Benchmark LogReg	0.9215	0.9172	0.9172
Benchmark SVM	0.9258	0.9204	0.9215

The second method gives significantly improved performance for each Proteobacteria data set, now beating even the logistic regression and SVM benchmarks. In each case, the network performed better with 500, rather than 5000, training epochs.

We next look at the data for the Euglenozoa sets. Again, 200 runs of 500 epochs and 50 runs of 5000 epochs were carried out, using a learning rate of 0.05, a momentum of 0.1, an initial b value of 0.5, and one hidden layer neuron. The results are shown in Table 13, with the best classification for each set listed in blue.

Table 13: Mean AUC Results Following for Euglenozoa Using 2nd Method. Benchmarks from [19]

	BLAST	SW	LA
95% CI, Method 2, 500 Iter.	(0.828056, 0.829626)	(0.863856, 0.864371)	0.945455 *
95% CI, Method 2, 5000 Iter.	(0.848539, 0.851098)	0.877273 *	(0.928243, 0.929575)
95% CI, Standard, 500 Iter.	(0.734328, 0.741763)	(0.771511, 0.780534)	(0.798451, 0.801685)
95% CI, Standard, 5000 Iter.	(0.752322, 0.761315)	(0.772611, 0.782844)	(0.792371, 0.799811)
Benchmark ANN	0.8318	0.8045	0.8182
Benchmark LogReg	0.7909	0.8227	0.8545
Benchmark SVM	0.8318	0.8182	0.8182

In this case, the second method with 5000 epochs performed best for the BLAST and Smith-Waterman data sets, while the second method with 500 epochs performed best on the local alignment kernel data set. For each set, all of the benchmark values were significantly beaten by either the short or long training sessions using the second method. An asterisk (*) indicates an AUC value that was identical across all training runs.

3.3 Discussion

When using the first method of an ideal low-training proportion, the new hidden layer activation function appears to improve network performance on certain data sets with noisy or possibly mislabeled data. However, this performance increase is not consistent across different data sets, and this method is not suggested as a universal replacement of the standard model. When using the second method of gradient ascent, one sees a significant increase in performance in the classification of 3PGK data over the standard model, the first new method, and available benchmarks for three different classifiers. One area of possible further research is to determine the characteristics of a data set that result in improved or decreased performance for these new models.

There are many other strategies employed to reduce overtraining in the presence of noisy, peculiar, or mislabeled data. These often include performing statistical analysis on data and validation set performance to determine which factors or patterns to exclude from training [7]. Future research could focus on comparison between this new activation function and other methods of overtraining prevention, both in terms of classification performance and computational requirement. In addition, the combination of statistical analysis and this activation function may provide further improved results.

Finally, improvements could be made on the algorithms used for updating the activation function. In this research, for the first method, a desired proportion of low-training data has been selected for classification problems based on testing and educated speculation. Further work could pinpoint optimal values for low-training data or implement a different algorithm altogether. One possibility would be the determination of an optimal setting for the b parameter, possibly via validation sets, followed by training on a static function. For the second method, gradient ascent was performed along the partial derivative with respect to b. Future work could implement various scaling factors to fine-tune this gradient ascent. The method could also be generalized to overcome overtraining in networks with a larger hidden layer, as such a method could expand the class of problems to which this method applies.

BIBLIOGRAPHY

[1] S. F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, Basic local alignment search tool, *J Mol Biol*, **215** (1990), 403-410.

[2] S. Belongie, J. Malik, and J. Puchiza, Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **24(4)**, (2002), 509522.

[3] T. Bylander, Universal Approximation, [`http://www.cs.utsa.edu/~bylander/cs4793/univ-approx.pdf`], (2003). Accessed May 2011.

[4] G. Cybenko, Approximation by Super-positions of a Sigmoid Function, *Mathematics of Control, Signals and Systems*, **2**(1989), 303-314.

[5] C. Gershensen, Artificial Neural Networks for Beginners, *CoRR*, cs.NE/0308031 (2003), available at [`http://arxiv.org/abs/cs/0308031v1`].

[6] G. Guo, A.K. Jain, W. Ma, and H. Zhang, Learning Similarity Measure for Natural Image Retrieval with Relevance Feedback, *IEEE Trans. Neural Networks*, **4** (2002), 811-820.

[7] P. Hartono, S. Hashimoto, Learning from imperfect data, *Applied Soft Computing*, **7** (2007), 353-363.

[8] T. Haynes, D. Knisley and J. Knisley, Using a Neural Network to Identify Secondary RNA Structures Quantified by Graphical Invariants, *Communications in Mathematical and in Computer Chemistry / MATCH* **60**, 277 (2008).

[9] J. Knisley, L. Glenn, K. Joplin and P. Carey, *Artificial Neural Networks for Data Mining and Feature Extraction*, in *Quantitative Medical Data Analysis Using Mathematical Tools and Statistical Techniques*, eds. I. Stojmenovic and A. Nayak (World Scientific, 2006).

[10] T. Lasko, J. Bhagwat, K. Zou and L. Ohno-Machado, The use of receiver operating characteristic curves in biomedical informatics, *Journal of Biomedical Informatics* **38**, 404 (2005).

[11] S. Lawrence, C. Giles and A. Tsoi, Lessons in Neural Network Training: Overfitting May be Harder than Expected, *Proceedings of the Fourteenth National Conference on Artificial Intelligence* **AAAI-97**, 540 (1997).

[12] C. May, Demonstrations of neural network computations involving students, *Journal of Undergraduate Neuroscience Education*, **8** (2010), A116-A121.

[13] J. McDonald, Handbook of Biological Statistics, [`http://udel.edu/~mcdonald/statintro.html`], (2009). Accessed June 2011.

[14] S. A. Nelson, An Introduction to Curve Fitting and Neural Nets, [`http://www.optimaldesign.org/Documents/nets.pdf`], (1997). Accessed December 2010.

[15] P. Neville, Decision Trees for Predictive Modeling, [http://bus.utk.edu/stat/datamining/DecisionTreesforPredictiveModeling(Neville).pdf], (1999). Accessed June 2011.

[16] G. Orr, Introduction to Neural Networks, [http://www.willamette.edu/~gorr/classes/cs449/intro.html], (1999). Accessed February 2011.

[17] M. Peniak, D. Marocco, and A. Cangelosi, Autonomous Robot Exploration of Unknown Terrain: A Preliminary Model of Mars Rover Robot, *ASTRA 2008*, Noordwijk, (2008).

[18] J.D. Pollack, Q. Li, and D.K. Pearl, Taxonomic utility of a phylogenetic analysis of phosphoglycerate kinase proteins of Archaea, Bacteria, and Eukaryota: insights by Bayesian analyses, *Mol Phylogenet Evol*, **35** (2005), 420-430.

[19] Protein Classification Benchmark Collection, [http://hydra.icgeb.trieste.it/benchmark/], (2006). Accessed March 2011.

[20] G. Qin and X. Zhou, Empirical Likelihood Inference for the Area Under the ROC Curve, *Biometrics*, **62**(2006), 613-622.

[21] P. Rice, I. Longden, and A. Bleasby, EMBOSS: the European Molecular Biology Open Software Suite, *Trends Genet*, **16** (2000), 276-277.

[22] H. Saigo, J.P. Vert, N. Ueda, and T. Akutsu, Protein homology detection using string alignment kernels, *Bioinformatics*, **20** (2004), 1682-1689.

[23] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, *J. Mol. Biol.*, **147** (1981), 195-197.

[24] The Insurance Company Benchmark (CoIL 2000), *The UCI KDD Archive*, available at [`http://kdd.ics.uci.edu/databases/tic/tic.html`].

[25] L. Wang, K. Smith, On chaotic simulated annealing, *IEEE Trans Neural Netw*, **9** (1998), 716-718.

[26] L. Weruaga, Multilayer Neural Networks, Beamer presentation, (2006).

[27] S. Xue, M. Yang, C. Li, and J. Nie, Meteorological Prediction Using Support Vector Regression with Genetic Algorithms, *ICISE 2009*, (2009), 4931-4935.

[28] D. Yu, Early Stopping of a Neural Network via a Receiver Operating Curve, M.S. Thesis, East Tennessee State University, 2010.

APPENDIX

Python Code

```
import random

import math

import numpy

import matplotlib.pyplot as plt


random.seed()


def rand(a, b):

    return (b-a)*random.random() + a


# Defines our original sigmoid function:

def sigmoid(x):

    try:

        y = math.exp(-x)

    except OverflowError:

        return 0.

    return 1/(1+y)


# Defines the new sigmoid function - goes from -1 to 1

def newsig(x,b):

    return (sigmoid(x+b)+sigmoid(x-b))-1.


# Can be used to normalize a list or array, if needed

def normalize(x):

    xmean = numpy.mean(x)

    xstd = numpy.std(x, ddof = 1)

    for i in len(x):

            x[i] = (x[i]-xmean)/xstd
```

```
# Class defining the neural network
class ANeuron:
    def __init__(self, ni, nh, no):
        # Number of in, hidden, out nodes set
        self.ni = ni + 1 # + 1 for bias
        self.nh = nh
        self.no = no

        # weights and activations activated
        self.wi = numpy.zeros((self.ni,self.nh))
        self.wo = numpy.zeros((self.nh,self.no))
        for i in xrange(self.ni):
            for j in xrange(self.nh):
                self.wi[i][j] = rand(-0.2,0.2)
        for i in xrange(self.nh):
            for j in xrange(self.no):
                self.wo[i][j] = rand(-0.2,0.2)
        self.ai = numpy.zeros(self.ni)
        self.ao = numpy.zeros(self.no)
        self.ah = numpy.zeros(self.nh)

        # This is in case the initialization weights are needed
        self.first_state = [self.wi.copy(),self.wo.copy()]

        # This defines the proportion and error bound for the
        # amount of "low-training" data.
        self.low_train_prop = 0.1
        self.low_train_tol = 0.02

    # The derivative for the new activation function.
    # is defined here
```

```
def newsigprime(self, x, b, k):
    return -self.ah[k]*(self.ah[k]+1)+2*(sigmoid(x+b)*sigmoid(x-b))

# This method randomizes the weights and
# activations for another training session.
def untrain(self):
    for i in xrange(self.ni):
        for j in xrange(self.nh):
            self.wi[i][j] = rand(-0.2,0.2)
    for i in xrange(self.nh):
        for j in xrange(self.no):
            self.wo[i][j] = rand(-0.2,0.2)
    self.ai = numpy.zeros(self.ni)
    self.ao = numpy.zeros(self.no)
    self.ah = numpy.zeros(self.nh)
    self.first_state = [self.wi.copy(),self.wo.copy()]

# This methoid performs ROC analysis, finding
# best threshold and AUC. Translated from Maple
# code. Note that it currently only handles the
# case of a network with a SINGLE output.
def roc_analysis(self, actual, predicted):
    pos_values = []
    neg_values = []
    for i in xrange(len(actual)):
        if actual[i] > 0.9:
            pos_values.append(predicted[i][0])
        else:
            neg_values.append(predicted[i][0])
    pos_values.sort()
    pos_values.reverse()
```

```
pos_num = len(pos_values)
neg_values.sort()
neg_values.reverse()
neg_num = len(neg_values)
threshs = []
for i in xrange(len(predicted)):
    threshs.append(predicted[i][0])
threshs = set(threshs)
threshs = list(threshs)
threshs.sort()

roc_list = [[1,1]]
best_thresh = [2,0,[1,1]]
for thresh in threshs:
    while len(pos_values) > 0 and pos_values[-1] <= thresh:
        pos_values.pop()
    while len(neg_values) > 0 and neg_values[-1] <= thresh:
        neg_values.pop()
    tmp = ([len(neg_values) / float(neg_num),
            len(pos_values) / float(pos_num)])
    distance = math.sqrt(tmp[0] ** 2 + (tmp[1]-1) ** 2)
    if distance < best_thresh[0]:
        best_thresh = [distance, thresh, tmp]
    roc_list.append(tmp)
auc_value = 0.
for i in xrange(len(roc_list)-1):
    auc_value += (abs(roc_list[i][0] - roc_list[i+1][0]) *
                  (roc_list[i][1] + roc_list[i+1][1]))
auc_value = auc_value / 2
return [best_thresh[1], auc_value]
```

```
# This is the feed-forward method. Takes an array of
# input plus the desired b value.
def forward(self, inputs, b):
    # To catch wrong number of inputs
    if len(inputs) != self.ni - 1:
        raise ValueError, 'Wrong # of inputs'

    # Sets activations, first input left as
    #   1 for bias.
    self.ai = numpy.insert(inputs,0,1.0)
    self.ao = numpy.zeros(self.no)
    self.ah = numpy.zeros(self.nh)

    # input_sums is inputs to hidden nodes,
    # hidden_sums is inputs to output node
    self.input_sums = numpy.zeros(self.nh)
    self.hidden_sums = numpy.zeros(self.no)

    # Calculates hidden node input, initializes
      # count of hidden layer activations in each
      # given range. These values could be changed.
    self.input_sums = numpy.dot(self.ai,self.wi)
    self.ah_count = [0.,0.,0.,0.,0.]

     # Calculates hidden activations, counts activations
     # for each range.
    for i in range(self.nh):
        self.ah[i] = newsig(self.input_sums[i], b)
        if self.ah[i] < -0.9:
            self.ah_count[0] += 1.
        elif self.ah[i] < -0.2:
            self.ah_count[1] += 1.
        elif self.ah[i] < 0.2:
            self.ah_count[2] += 1.
        elif self.ah[i] < 0.9:
            self.ah_count[3] += 1.
        else:
            self.ah_count[4] += 1.

    # Calculates outer layer input and then
    # the network output.
    self.hidden_sums = numpy.dot(self.ah,self.wo)
    for i in xrange(len(self.ao)):
        self.ao[i] = sigmoid(self.hidden_sums[i])

    # Returns output array
    return self.ao

def backpropagate(self, patterns, N, learn_rate, momentum, val_pats = [],
                  b_set = 0, modified_sigmoid = True):
    # N is the number of training epochs, val_pats should be a list
      # of validation patterns, if a validation set is used for early
      # stopping. b_set is the starting value for b in the
      # modified sigmoid model, and modified_sigmoid is True /
      # False based on whether the sigmoid will change during
      # training.

# Patterns should be input as lists where each
# list entry is [[inputs],[outputs]]
    anneal_std = learn_rate / 5.
    validation = False
    num_pats = len(patterns)
```

```
num_val_pats = len(val_pats)
self.xpat = numpy.zeros((num_pats, self.ni - 1))
self.ypat = numpy.zeros((num_pats, self.no))
for i in range(num_pats):
    self.xpat[i] = patterns[i][0]
    # outputs are re-sized to range from 0.005 to .995
    self.ypat[i] = patterns[i][1]
    self.ypat[i] = 0.99 * self.ypat[i] + 0.01 * 0.5
rand_list = range(len(patterns))
self.val_errors = numpy.zeros(N)
# This executes if using a validation set
if val_pats != []:
    validation = True
    self.val_x = numpy.zeros((num_val_pats, self.ni - 1))
    self.val_y = numpy.zeros((num_val_pats, self.no))
    val_output = numpy.zeros((num_val_pats, self.no))
    val_error_min = 0.
    self.best_state = []
    self.best_iter = 0
    for i in range(num_val_pats):
        self.val_x[i] = val_pats[i][0]
        self.val_y[i] = val_pats[i][1]
        self.val_y[i] = 0.99 * self.val_y[i] + 0.01 * 0.5

# Deltas are partial derivative values,
#    y_product and h_product are used
#    during backpropagation calcs.
self.delta_out = numpy.zeros((self.nh, self.no))
self.delta_in = numpy.zeros((self.ni,self.nh))
y_product = numpy.zeros((1, self.no))
h_product = numpy.zeros((1, self.nh))
```

```
    self.errors = numpy.zeros(N)
# self.b is the inflection point for the sigmoid
    self.b = b_set
    self.last_iter = 0
    self.best_iter = 0
    self.early_stopped = False
# matrices are used to (hopefully) speed up calculations,
# changexx are used to calculate change in weights,
# used for momentum
    self.change_wi = numpy.zeros((self.ni, self.nh))
    self.change_wo = numpy.zeros((self.nh, self.no))
    # This initializes matrices for input and output
    # weight annealing.
    anneal_in = numpy.zeros((self.ni,self.nh))
    anneal_out = numpy.zeros((self.nh, self.no))
    # This initializes proportion counts for the ranges
    # of hidden activations. Can be used in graphing
    # later to help visualize where the inputs are going.
    self.hdnpctlow = []
    self.hdnpctmidlow =[]
    self.hdnpctmid = []
    self.hdnpctmidhi = []
    self.hdnpcthi = []
    self.b_list = []
    # Backpropagation begins here. There is a bit of ugly
    # fiddling with matrices. Could maybe be implemented in a
    # faster way.
    for i in xrange(N):
        self.last_iter = i
        self.delta_in = numpy.zeros((self.ni, self.nh))
        self.delta_out = numpy.zeros((self.nh, self.no))
```

```
hdnlow = 0.
hdnmidlow = 0.
hdnmid = 0.
hdnmidhi = 0.
hdnhi = 0.
self.b_list.append(self.b)
for j in rand_list:     #rand_list used to randomize pats
    self.forward(self.xpat[j], self.b)
    hdnlow += self.ah_count[0]
    hdnmidlow += self.ah_count[1]
    hdnmid += self.ah_count[2]
    hdnmidhi += self.ah_count[3]
    hdnhi += self.ah_count[4]
    bpartial = 0.
    for k in xrange(self.no):
        y_product[0][k] = ((self.ypat[j][k] - self.ao[k]) *
                          self.ao[k] * (1 - self.ao[k]))
        self.errors[i] += 0.5 * (self.ypat[j][k] - self.ao[k]) ** 2
    # Partial with respect to hidden -> output weights
    # is calculated here.
    self.delta_out = numpy.dot(self.ah.reshape(self.nh, 1),
                               y_product)
    h_product[0] = numpy.dot(y_product, self.wo.T)
    for k in xrange(self.nh):
        h_product[0][k] = (h_product[0][k] *
                         self.newsigprime(self.input_sums[k],
                                          self.b, k))
    # Partial with respect to input -> hidden weights
    # is calculated here.
    for k in xrange(self.no):
        for m in xrange(self.nh):
            bpartial += (y_product[0][k] * self.wo[m][k] *
                         self.ah[m] *
                         (sigmoid(self.input_sums[m] + self.b) -
                          sigmoid(self.input_sums[m]-self.b)))
        self.delta_in = numpy.dot(self.ai.reshape(self.ni, 1),
                                  h_product)
        self.change_wi = (learn_rate * self.delta_in +
                          momentum * self.change_wi)
        self.change_wo = (learn_rate * self.delta_out +
                          momentum * self.change_wo)
        self.wi += self.change_wi
        self.wo += self.change_wo
        self.b -= (learn_rate * bpartial)
    random.shuffle(rand_list)
    # Proportions of hidden activation outputs in each range
    # finally calculated here
    self.hdnpctlow.append(hdnlow / (self.nh * len(patterns)))
    self.hdnpctmidlow.append(hdnmidlow / (self.nh * len(patterns)))
    self.hdnpctmid.append(hdnmid / (self.nh * len(patterns)))
    self.hdnpctmidhi.append(hdnmidhi / (self.nh * len(patterns)))
    self.hdnpcthi.append(hdnhi / (self.nh * len(patterns)))
    # The following is the algorithm for early stopping.
    # If a minimum in validation error is reached and
    # not lowered in the succeeding 50 iterations,
    # backpropagation is stopped and the network returns
    # to its state at minimum validation error.
    if validation:
        for j in xrange(num_val_pats):
            val_output[j] = self.forward(self.val_x[j], self.b)
            for k in xrange(self.no):
                self.val_errors[i] += 0.5 * (self.val_y[j][k] -
                                            val_output[j][k]) ** 2
                if i == 0 or self.val_errors[i] <= val_error_min:
                    val_error_min = self.val_errors[i]
                    stop_count = 0
                    temp_save = [self.wi.copy(), self.wo.copy(), self.b, i]
                else:
                    stop_count += 1
                    if stop_count == 50 or i == N - 1:
                        self.wi = temp_save[0]
                        self.wo = temp_save[1]
                        self.b = temp_save[2]
                        self.best_iter = temp_save[3]
                        self.early_stopped = True
                        return None
            # Occasional errors are printed to help get a feel for
            # network performance.
            if i%20==0:
                print self.errors[i]
            # If the modified network is being used,
            # and low-training proportion is too low or too high,
            # the b value is adjusted here.
            # Low proportion = increase b,
            # High proportion = decrease b.
            # The following is commented out as it pertains only
            # to the first method. The second method does not use it.
#           if modified_sigmoid == True:
#               if self.hdnpctmid[-1] < (self.low_train_prop -
#                                        self.low_train_tol):
#                   self.b += 0.05
#               elif self.hdnpctmid[-1] > (self.low_train_prop +
#                                          self.low_train_tol):
```

```
#                     self.b = max(self.b - 0.05, 0.)
            # Annealing done every 20th iteration.
            # These values could possibly use tweaking.
            if i%20==0 and i > 0:
                anneal_tol = self.errors[i] / 10000.
                if abs(self.errors[i] - self.errors[i-1]) < anneal_tol:
                    # annealing normally distributed around 0,
                    # st dev = annealSize
                    for j in xrange(self.ni):
                        for k in xrange(self.nh):
                            anneal_in[j][k] = random.gauss(0, anneal_std)
                            for m in xrange(self.no):
                                anneal_out[k][m] = random.gauss(0, anneal_std)
                    # annealing scaled by current error size
                    anneal_in = self.errors[i] * anneal_in
                    anneal_out = self.errors[i] * anneal_out
                    # change in weights updated w/ annealing
                    self.wi += anneal_in
                    self.wo += anneal_out
                    print 'Annealing now.'

    def createpatterns(self, num_pats, validation = False, num_val_pats = 0,
                       from_file = False, filename = '', sep_files = 'noinput',
                       train_file = '', test_file = '', type = 'x3x4'):
        # This method generates patterns for the
        # network. If not from a file, they will be generated
        # due to the "modified inclusive OR" rules.
        # Data files should have one pattern per row with outputs
        # at the end of the row. If one file is used,
        # training and evaluation patterns are selected
        # at random. This was just a convenience for me.
```

```
train_pats = []
test_pats = []
if validation:
    val_pats = []
    if num_val_pats == 0:
        num_val_pats = int(raw_input('Number of stopping pats? '))
if from_file:
    if sep_files == 'noinput':
        sep_files = str(raw_input('Training/testing data in separate files? '))
    if sep_files == 'yes':
        if train_file == '':
            train_file = str(raw_input('Training data from which file? '))
        if test_file == '':
            test_file = str(raw_input('Testing data from which file? '))
        file1 = open(train_file)
        for line in file1:
            line2 = []
            line1 = map(float, line.split())
            if len(line1) != (self.ni - 1) + self.no:
                raise ValueError, 'Incorrect # of inputs/outputs'
            for i in xrange(self.no):
                line2.append(line1.pop())
            line2.reverse()
            train_pats.append([line1,line2])
        if validation:
                random.shuffle(train_pats)
                for i in xrange(num_val_pats):
                    val_pats.append(train_pats.pop())
        file1.close()
        file2 = open(test_file)
        for line in file2:
```

```
        line2 = []
        line1 = map(float, line.split())
        if len(line1) != (self.ni - 1) + self.no:
            raise ValueError, 'Incorrect # of inputs/outputs'
        for i in xrange(self.no):
            line2.append(line1.pop())
        line2.reverse()
        test_pats.append([line1,line2])
    file2.close()
else:
    filename1 = filename
    if filename1 == '':
        filename1 = str(raw_input('Name of data file? '))
    all_pats = []
    file1 = open(filename1)
    for line in file1:
        line2 = []
        line1 = map(float, line.split())
        if len(line1) != (self.ni - 1) + self.no:
            raise ValueError, 'Incorrect # of inputs/outputs'
        for i in xrange(self.no):
            line2.append(line1.pop())
        line2.reverse()
        all_pats.append([line1,line2])
    file1.close()
    if validation:
        random.shuffle(all_pats)
        for i in xrange(num_val_pats):
            val_pats.append(all_pats.pop())
    train_pats = random.sample(all_pats,num_pats)
    test_pats = random.sample(all_pats,num_pats)
```

```
            del all_pats
    elif type == 'x3x4':
        for i in xrange(num_pats):
            train_pats.append([[],[0.]])
            test_pats.append([[],[0.]])
            for j in xrange(self.ni-1):
                train_pats[i][0].append(random.gauss(0, 10))
                test_pats[i][0].append(random.gauss(0, 10))
            if min(train_pats[i][0][2], train_pats[i][0][3]) > 0:
                train_pats[i][1][0] = 1
            elif max(train_pats[i][0][2],train_pats[i][0][3]) > 0:
                out_rand = random.random()
                # Mislabels, on average, 5% of data
                if out_rand < 0.9:
                    train_pats[i][1][0] = 1.
            if max(test_pats[i][0][2],test_pats[i][0][3]) > 0:
                test_pats[i][1][0] = 1.
        if validation:
            for i in xrange(num_val_pats):
                val_pats.append([[],[0]])
                for j in xrange(self.ni-1):
                    val_pats[i][0].append(random.gauss(0,10))
                if min(val_pats[i][0][2],val_pats[i][0][3]) > 0:
                    val_pats[i][1][0] = 1.
                elif max(val_pats[i][0][2],val_pats[i][0][3]) > 0:
                    out_rand = random.random()
                    if out_rand < 0.9:
                        val_pats[i][1][0] = 1.
    pats = [train_pats, test_pats]
    if validation:
        pats.append(val_pats)
    # The method returns [train_pats, test_pats(, validation_pats)]
    return pats



def sampled_train_set(self,pats, set_choice, random_sample_pos_num,
                      random_sample_neg_num):
    # This method randomly samples a set of negative
    # patterns from the training set. This was used
    # for convenience while studying TIC data.
    all_pos_pats = []
    pos_pats = []
    all_neg_pats = []
    neg_pats = []
    if set_choice == 0:
        sample_pos = True
        sample_neg = False
    elif set_choice == 1:
        sample_pos = False
        sample_neg = True
    else:
        sample_pos = True
        sample_neg = True
    for pattern in pats:
        if pattern[1][0] > 0.9:
            all_pos_pats.append(pattern)
        else:
            all_neg_pats.append(pattern)
    if sample_pos:
        random.shuffle(all_pos_pats)
    else:
        random_sample_pos_num = len(all_pos_pats)
    if sample_neg:
        random.shuffle(all_neg_pats)
    else:
        random_sample_neg_num = len(all_neg_pats)
    for i in xrange(random_sample_neg_num):
        neg_pats.append(all_neg_pats.pop())
    for i in xrange(random_sample_pos_num):
        pos_pats.append(all_pos_pats.pop())
    all_pats = []
    for pattern in pos_pats:
        all_pats.append(pattern)
    for pattern in neg_pats:
        all_pats.append(pattern)
    return all_pats

def mislabel(self,pats,mislabeled_num):
    # This method randomly mislabels a given number of
    # training patterns. It is designed for patterns
    # with only ONE output and a binary (0,1) classification.
    mislabeled = random.sample(xrange(len(pats)), mislabeled_num)
    for item in mislabeled:
        pats[item][1][0] = 1 - pats[item][1][0]
    return pats

def stat_analysis(self, num_sessions, num_pats, validation, from_file, N,
                  learn_rate, momentum, low_train_prop, low_train_tol, b_set):
    # This method is run by another program to
    # generate AUC data and graphs for comparison with
    # old sigmoid function. User input is very specific,
    # as it was designed for my personal use.
    filename1 = str(raw_input('Filename for new sigmoid data? '))
```

```
filename2 = str(raw_input('Filename for old sigmoid data? '))
filename4 = str(raw_input('Filename for test set output? '))
fig_prefix = str(raw_input('Prefix for error figures? '))
file1 = open(filename1,'a')
file1.write('New Sigmoid Data\nAUCTrain\tAUCTest\t\tFinal Err.\t' +
            'Starting B\tFinal B\t\tL. Rate\t\tMomentum\t' +
            'Iterations\n\n')
file1.close()
file2 = open(filename2,'a')
file2.write('Old Sigmoid Data\nAUCTrain\tAUCTest\t\tFinal Err.\t' +
           'Final B\t\tL. Rate\t\tMomentum\tIterations\n\n')
file2.close()
filename3 = ''
train_file = ''
test_file = ''
sep_files = ''
num_val_pats = 0
if validation:
    num_val_pats = int(raw_input('How many validation patterns? '))
random_sample = False
random_sample_input = str(raw_input('Randomly sample patterns? '+
                                     'Type yes or no. '))
mislabeled_value = False
mislabeled_input = str(raw_input('Mislabel training patterns? ' +
                                  'Type yes or no. '))
if mislabeled_input == 'yes':
    mislabeled_value = True
    mislabeled_num = int(raw_input('Mislabel how many training patterns? '))
if random_sample_input == 'yes':
    set_choice = int(raw_input('Type 0 to sample pos, 1 for neg, 2 for both. '))
    random_sample = True
```

```
        if set_choice == 0:
            random_sample_pos_num = int(raw_input('Pos random sample size? '))
            random_sample_neg_num = 0
        elif set_choice == 1:
            random_sample_neg_num = int(raw_input('Neg Random sample size? '))
            random_sample_pos_num = 0
        else:
            random_sample_pos_num = int(raw_input('Pos random sample size? '))
            random_sample_neg_num = int(raw_input('Neg Random sample size? '))
    if from_file:
        sep_files = str(raw_input('Training/testing data in separate ' +
                                  'files? Type yes or no. '))
        if sep_files == 'yes':
            train_file = str(raw_input('Training data from which file? '))
            test_file = str(raw_input('Testing data from which file? '))
        else:
            filename3 = str(raw_input('Read data from which file? '))
    pats = []
    #self.low_train_prop = low_train_prop
    #self.low_train_tol = low_train_tol
    for i in range(num_sessions):
        self.untrain()
        pats_master = self.createpatterns(num_pats, validation, num_val_pats,
                                          from_file,filename3, sep_files,
                                          train_file,test_file)
        pats = pats_master[:]
        val_pats = []
        if validation:
            val_pats = pats.pop()
        self.test_pats = pats.pop()
        all_train_pats = pats.pop()
```

```
train_pats = all_train_pats
if random_sample:
    train_pats = self.sampled_train_set(all_train_pats, set_choice,
                                        random_sample_pos_num,
                                        random_sample_neg_num)
if mislabeled_value:
    train_pats = self.mislabel(all_train_pats, mislabeled_num)
train_actual = []
test_actual = []
train_predict = []
test_predict = []
self.untrain()
self.backpropagate(train_pats, N, learn_rate, momentum, val_pats,
                   b_set, True)
plt.figure()
plt.subplot(2,1,1)
plt.plot(xrange(len(self.errors)), self.errors, 'r-',
         xrange(len(self.val_errors)), self.val_errors, 'b-',
         xrange(len(self.b_list)),self.b_list,'g-')
fig_name = ('%s' %fig_prefix + 'nsError' + 'iter%s' %i +
           'b%s' %b_set + '.png')
#plt.savefig(figname)
plt.subplot(2,1,2)
plt.plot(xrange(len(self.errors)), self.errors, 'r-',
         xrange(len(self.val_errors)), self.val_errors, 'b-')
plt.plot(xrange(len(self.hdnpctlow)), self.hdnpctlow, color='cyan',
         linestyle='solid')
plt.plot(xrange(len(self.hdnpctlow)), self.hdnpctmidlow,
         color='green', linestyle='solid')
plt.plot(xrange(len(self.hdnpctlow)), self.hdnpctmid,
         color='yellow', linestyle='solid')
```

```
plt.plot(xrange(len(self.hdnpctlow)), self.hdnpctmidhi,
         color='orange', linestyle='solid')
plt.plot(xrange(len(self.hdnpctlow)), self.hdnpcthi,
         color='magenta', linestyle='solid')
plt.ylim((0,1))
#figname = '%s' %figprefix + 'nsErrorShort' + 'iter%s' %i + 'b%s' %k + '.png'
plt.savefig(fig_name)
plt.close()
for j in xrange(len(train_pats)):
    train_predict.append(self.forward(train_pats[j][0], self.b))
    train_actual.append(train_pats[j][1][0])
file4 = open(filename4,'a')
for j in xrange(len(self.test_pats)):
    test_predict.append(self.forward(self.test_pats[j][0], self.b))
    file4.write('%f\t' %(test_predict[j]))
    test_actual.append(self.test_pats[j][1][0])
file4.write('\n')
file4.close()
train_auc = self.roc_analysis(train_actual,train_predict)
test_auc = self.roc_analysis(test_actual,test_predict)
if validation:
    self.last_iter = self.best_iter
last_error = self.errors[self.last_iter]
file1 = open(filename1,'a')
file1.write('%f\t%f\t%f\t%f\t%f\t%f\t%f\t%i\n'
            %(train_auc[1], test_auc[1], last_error, b_set,
              self.b, learn_rate, momentum, self.last_iter + 1))
file1.close()
self.untrain()
print i+1,'newsig dataset(s) collected'
train_actual = []
```

```
test_actual = []
train_predict = []
test_predict = []
self.backpropagate(train_pats, N, learn_rate, momentum, val_pats,
                   0., False)
plt.figure()
plt.subplot(2,1,1)
plt.plot(xrange(len(self.errors)), self.errors, 'r-',
         xrange(len(self.val_errors)), self.val_errors, 'b-')
fig_name = '%s' %fig_prefix + 'osError' + 'iter%s' %i + '.png'
#plt.savefig(figname)
plt.subplot(2,1,2)
plt.plot(xrange(len(self.errors)), self.errors, 'r-',
         xrange(len(self.val_errors)), self.val_errors,'b-')
plt.ylim((0,1))
#figname = '%s' %figprefix + 'osErrorShort' + 'iter%s' %i + '.png'
plt.savefig(fig_name)
plt.close()
for j in xrange(len(train_pats)):
    train_predict.append(self.forward(train_pats[j][0], self.b))
    train_actual.append(train_pats[j][1][0])
for j in xrange(len(self.test_pats)):
    test_predict.append(self.forward(self.test_pats[j][0], self.b))
    test_actual.append(self.test_pats[j][1][0])
train_auc = self.roc_analysis(train_actual, train_predict)
test_auc = self.roc_analysis(test_actual, test_predict)
if validation:
    self.last_iter = self.best_iter
last_error = self.errors[self.last_iter]
file2 = open(filename2,'a')
file2.write('%f\t%f\t%f\t%f\t%f\t%f\t%i\n'
            %(train_auc[1], test_auc[1], last_error, self.b,
              learn_rate, momentum, self.last_iter + 1))
file2.close()
self.untrain()
print i+1,'oldsig dataset(s) collected'
```

VITA

JEFF BONNELL

Education:	B.S. Mathematics, East Tennessee State University, Johnson City, Tennessee 2008
	M.S. Mathematical Sciences, East Tennessee State University, Johnson City, Tennessee 2011
Professional Experience:	Graduate Assistant, Tutor, East Tennessee State University, Johnson City, Tennessee, 2009–2010
	Graduate Teaching Assistant, East Tennessee State University, Johnson City, Tennessee, 2010–2011
Awards:	Outstanding Graduate Student in Mathematics, Department of Mathematics and Statistics, East Tennessee State University, 2011