Artificial Neural Networks
Mechanics and Implementations

AN INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS
WITH MATLAB IMPLEMENTATIONS

SPRING 2015

COMPILED BY
LEONARD STRNAD

ADVISED BY
DR. JONATHAN SCOTT

Cleveland State University

Contents

1 Introduction **3**

2 Single-Layer Networks **3**

2.1 The 2-Category Problem: 4

2.2 The Perceptron: 4

2.2.1 The Perceptron Criterion 5

2.2.2 Gradient descent 6

3 Multi-Layer Networks **6**

3.1 Back-Propagation 7

3.1.1 Example using the hyperbolic tangent activation function 9

4 Learning and Generalization **10**

4.1 Bias and Variance 12

5 Example Data and Matlab Implementation **13**

5.1 Simple Linear Classification 13

5.2 Nonlinear Classification 14

5.3 Nested Data 14

5.4 Multiple Classes 15

6 Conclusion **15**

7 Matlab Code **16**

7.1 One Hidden Layer Network 16

7.2 Two Hidden Layer Network 18

7.3 Three Hidden Layer Network 20

Abstract. *Artificial Neural Networks are powerful in that they have the ability to learn any continuous function or underlying distribution. The architecture of the system allows for very dynamic training of the network parameters. The training of networks can 'easily' be implemented in embedded systems, assigned to the GPU, or trained on a personal computer. Although Neural Networks strongly rely on computational power, cloud computing and scalable systems will be able to facilitate very complex networks. This paper simply explores the fundamentals of Artificial Neural Networks, highlights applications and implementations, and hopefully motivates investigation.*

1. Introduction

An Artificial Neural Network is a directed graph where each node operates on a sum of some weighted inputs, computes a value with an activation function and outputs a single value. Proper structure and training regime will solve for weights which tune the network to learn the underlying distribution of the given data on which the network was trained. If properly constructed, the network can be extremely robust to odd distributions and can operate very fast on new examples. There are many implementations of these in various contexts e.g. computer vision, temporal/sequential data, etc. This paper simply explores the supervised implementation of neural networks; however, there are unsupervised learning and reinforcement learning uses of the neural network [8], [10]. The paper will walk through the simple perceptron, multiply layer neural network, the learning regime, generalization, and look at a few example data sets.

2. Single-Layer Networks

A single-layer Neural Network (NN) can be thought of as a linear discriminant function where the function consists of as a transformed linear combination of input variables where the coefficients or weights are parameters of the model. One can solve for the coefficients using least-squares, Fisher discriminant, or perceptron learning methods. A single-layer network with a threshold output unit has a decision boundary which is a hyperplane. Understanding these single-layer NNs can help in understanding the multiple-layer NN.

2.1. The 2-Category Problem:

A 2-category classification problem essentially searches for a linear discriminant function that can be written as

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{1}$$

where the sign of $y(\boldsymbol{x})$ determines the class of $\boldsymbol{x}$ which is the feature/input vector, $\boldsymbol{w}$ is the *d-dimensional* weight vector, and $-w_0$ is the threshold. If we let $\boldsymbol{x} = (1, \boldsymbol{x})$ and $\boldsymbol{w} = (w_0, \boldsymbol{w})$, then the decision boundary can be seen as $y(\boldsymbol{x}) = 0$ which is a *d-dimensional* hyperplane passing through the origin. This linear discriminant function can be represented with a single layer NN (Figure 1) and simplified as follows:

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} \tag{2}$$

Linear discriminants for multiple classes create decision regions and can be shifted to the origin of similar form

$$y_k(\boldsymbol{x}) = {\boldsymbol{w}_k}^T \boldsymbol{x} = \sum_{i=1}^{d} w_{ki} x_i \tag{3}$$

We can transform the input vector $\boldsymbol{x}$ to generalize the discriminant functions and allow for a larger class of functions $y_k(\boldsymbol{x})$. They can be non-linear and are called *basis functions* or *activation functions*, denoted $\phi_j(\boldsymbol{x})$. Similarly, we can absorb biases by constructing $\phi_0(\boldsymbol{x}) = 1$ and get the form

$$y_k(\boldsymbol{x}) = \sum_{j=0}^{M} w_{ki} \phi_j(x) \tag{4}$$

Here, M, is the dimension of the input vector. The basis function chosen is often the same for all values of k. It is typically the sigmoid or hyperbolic-tangent function. It is important to note that single-layer Networks can only represent data that is *linearly separable*. Least-squares techniques can be used to solve for the weights which minimize the error. Essentially, the projection of the vector of target values onto the subspace spanned by the basis vectors constructed by $\phi(\boldsymbol{x})$ minimizes the distance between the predicted and target value.

2.2. The Perceptron:

Perceptrons are single-layer networks with threshold activation functions. They were introduced as applications of classification problems. Block (1962) provides a review of the perceptron properties [2]. The activation function operates on the sum of the weight multiplied with the basis function for each dimension of the input vector. Similarly, if we set ϕ_0 to one along with the corresponding bias parameter $w_)$, the the output of the perceptron is

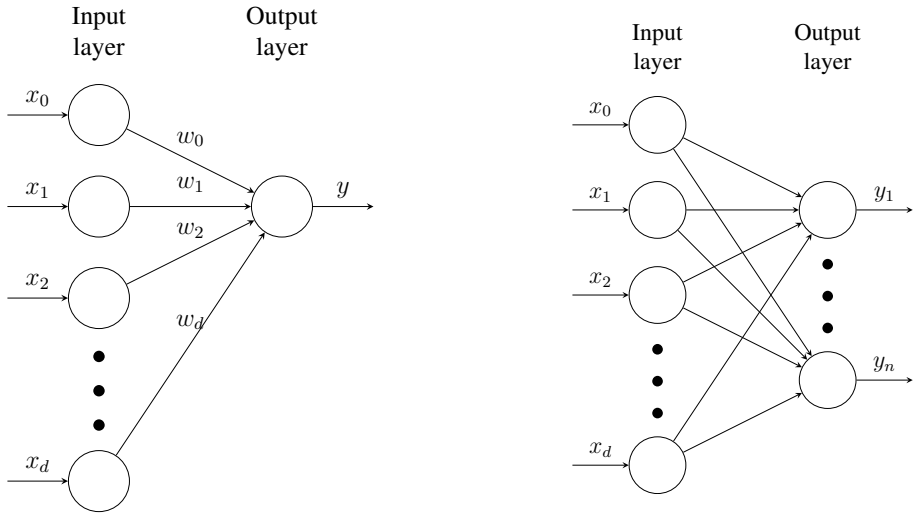

Figure 1: (Left) A single-layer Neural Network representation of a Linear Discriminant function. (Right) A single-layer Neural Network representation of multiple linear discriminant function.

$$y = g\left(\sum_{j=0}^{M} w_j \phi_j(x)\right) = g(\boldsymbol{w}^T \boldsymbol{\phi}) \tag{5}$$

.

M is the dimension of the input vector if for every attribute of the input vector there is one corresponding activation function ϕ_i. M may also be the count of activation functions which operate on a subset of attributes of the input vector.

2.2.1. The Perceptron Criterion

The perceptron criterion provides an error function of the weights. If $t = 1$ and $\boldsymbol{w}^T \boldsymbol{\phi} > 0$ denotes the target value and output belong to class C_1 and $t = -1$ and $\boldsymbol{w}^T \boldsymbol{\phi} < 0$ denotes the target and output belong to class C_2, then we want $\boldsymbol{w}^T(\boldsymbol{\phi t}) > 0$. Bishop (1995) suggests the corresponding error should be

$$E(\boldsymbol{w}) = -\sum \boldsymbol{w}^T(\boldsymbol{\phi}(\boldsymbol{x})t) \tag{6}$$

for some feature vector x and its corresponding target value, t [1]. We only sum the vectors $\boldsymbol{\phi}(\boldsymbol{x})$ which are misclassified. This function is proportional to the sum of distances of misclassified feature vectors from the decision boundary. If the target does not match, then $E(\boldsymbol{w})$ will be increasing monotonic. Since E is a differentiable function of $\boldsymbol{w}$, we can minimize using the iterative gradient descent algorithm.

2.2.2. Gradient descent

The gradient descent algorithm is an iterative technique which starts in a random state and then moves in the direction of steepest descent if we add $-\eta\nabla_{\boldsymbol{w}}E$ to the previous state. η is the learning rate parameter. The perceptron criterion provides an error function that is differentiable. Therefore, we can train the perceptron using

$$\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} + \eta\boldsymbol{\phi}(\boldsymbol{x})t \tag{7}$$

The algorithm goes through each feature vector, $\boldsymbol{x}$, and adjusts the weights if the feature vector is misclassified. It is pretty clear to see that $\eta\boldsymbol{\phi}(\boldsymbol{x})t$ will decrease as τ increases. Assuming the data is linearly separable there is an interesting result. The *perceptron convergence theorem* is interesting in that the gradient descent rule is guaranteed to find a solution in a finite number of steps [5]. However, there are many non-linearly separable data sets. This motivates the use of the multi-layer perceptron (MLP).

3. Multi-Layer Networks

Multiple layer Neural Networks have hidden nodes between the input and output nodes. Networks with just one hidden layer are capable of approximating any continuous function [1]. An example of the network topology is illustrated in Figure 2. Actually, Bishop (1995) reports that general networks with a sufficient number of hidden layers can approximate any functional continuous mapping from one finite dimensional space to another.

Each hidden layer is obtained by transforming the linear sum of weights *times* corresponding entry in feature vector. The outputs of the network are similarly obtained by transforming sum of the hidden activation layers which lead to that output. If we absorb the bias for the input vector in x_0 and the bias in H_0, then we can express the activation for the hidden layer as

$$a_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i \tag{8}$$

and the activation of hidden unit j is then obtained by using an activation function g to give

$$z_j = g(a_j) \tag{9}$$

Similarly, the output, y_k, is obtained by transforming the linear combination of hidden units with a non-linear activation function, $g*$, as follows

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \tag{10}$$

and using the activation function, $g^*(\cdot)$, we have the output

$$y_k = g^*(a_k) \tag{11}$$

The weights ${w_{kj}}^{(1)}$ and ${w_{kj}}^{(2)}$ correspond the the first and second layer between the j th and k th node. The idea extends as the number of hidden layers increases. For the

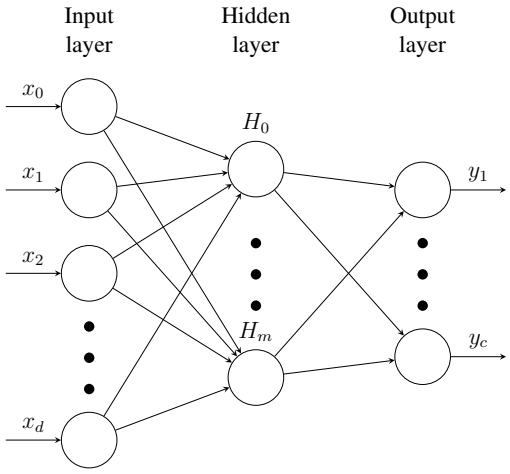

Figure 2: An example of a neural network with one hidden layer. There are also two layers of weights. One between input and hidden and another between hidden and output.

case of the two-layer network, the complete equation for y_k is

$$y_k = g^* \left(\sum_{j=0}^{M} w_{kj}^{(2)} g \left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i) \right) \right) \tag{12}$$

In order to get a better understanding of the role of the hidden threshold unit, we can think of each activation at each hidden unit or output unit as a decision boundary or composition of decision boundaries. The single-layer neural network with a threshold output unit creates a decision boundary which is a hyperplane. A neural network with two layers of hidden weights computes a hyperplane–decision boundary–for each hidden unit and the output threshold function computes a logical AND. This essentially can construct a convex boundary region. Networks with three hidden layers are able to construct non-convex and disjoint decision regions [6].

3.1. Back-Propagation

In order to solve for the weights which minimize the error, we will consider the most common technique used–*error back-propagation*. Essentially, if we consider differentiable activation functions, we can use the chain rule to calculate the derivative of the error w.r.t. the weights and update according to a similar iterative gradient descent approach discussed above. In order to define a method for general network topologies, x_i (input) can be considered as a_i and y_k (output) can be considered as z_k according to

$$a_j = \sum_{i=0}^{d} w_{ji} z_i \tag{13}$$

and

$$z_j = g(a_j) \tag{14}$$

The error function for each pattern is often the standard sum-of-squares error function (difference between target, t_k, and predicted, y_k) for each output c. The total is the sum of errors from each n input vector :

$$E = \sum_n E^n = \sum_n \frac{1}{2} \sum_{k=1}^{c} (y_k - t_k)^2 \tag{15}$$

The derivative of the error w.r.t. w_{ji} is computed using the chain rule:

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{16}$$

Evaluating $\frac{\partial E^n}{\partial a_k}$ for the output unit is straight forward. Recall the output, y_k, and z_k are considered as the same. Thus,

$$\frac{\partial E^n}{\partial a_k} = \frac{\partial E^n}{\partial y_k} \frac{\partial y_k}{\partial a_k} = \frac{\partial E^n}{\partial y_k} g'(a_k) = (y_k - t_k) g'(a_k) \tag{17}$$

Now, we need to find an expression for $\frac{\partial E^n}{\partial a_j}$ when a_j is not the output unit. We can use the chain rule keeping in mind all the k hidden or output units j send a connection. Thus,

$$\frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{18}$$

Now, if we define $\frac{\partial E^n}{\partial a_k} = \delta_k$, and recognize that $\frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial z_i} \frac{\partial z_i}{\partial a_j} = w_{kj} g'(a_j)$, then we can substitute δ_k into $\frac{\partial E^n}{\partial a_j}$ and get a recursive function for $\frac{\partial E^n}{\partial a_j}$ which yields

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k \tag{19}$$

Furthermore, $\frac{\partial a_j}{\partial w_{ji}}$ is simply

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{20}$$

Finally, we have a recursive expression for the total error over all instances of the data set

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j z_j \tag{21}$$

where j indexes any hidden unit of nodes and would be substituted with k for an output unit of nodes. For simplicity, we only discuss the same activation function, $g(\cdot)$, across the network.

So, if we (1) initialize weights to some small random value and forward a pattern vector through the network using differentiable activation functions, (2) evaluate δ_k (the output values of $\frac{\partial E^n}{\partial a_k}$), (3) back-propagate using the recursive function defined above, then (4) we can use $\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$ to evaluate the error w.r.t. each weight for each pattern vector.

Now, to find the total error we sum the error for each input vector and get

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^n}{\partial w_{ji}} = \sum_k \delta_j z_j \tag{22}$$

which then allows us to use the gradient descent algorithm to update each layer of weights after each pass of a input vector according to

$$w_{ji}^{\tau+1} = w_{ji}^{\tau} + \eta\delta_j x_i \tag{23}$$

for the first layer,

$$w_{kj}^{\tau+1} = w_{kj}^{\tau} + \eta\delta_k z_j \tag{24}$$

for the second layer and so on.

3.1.1. Example using the hyperbolic tangent activation function

Lets try to illustrate the back-propagation procedure with a neural network with just two layers of weights i.e. one hidden unit and one output unit. This structure is illustrated in Figure 2 with only one output node. We will use the hyperbolic tangent function as the activation function at the hidden units. For the output activation will use a piecewise linear function $g(\cdot)$ that computes the logical AND. That is, if the number of hidden layers is M, and the bias on the output is set to $-M$, then the output unit will output 1 iff every hidden unit has an output of 1. The hyperbolic tangent function is useful for classification problems because it will output ± 1 depending on the sign of the sum of the hidden units.

First, lets find the activation functions for the hidden units. These are found by summing the linear combination of weights and input components and transforming with the hyperbolic tangent activation function. That is

$$a_j = \sum_i w_{ji} x_i \tag{25}$$

gets transformed by $tanh(a_j)$ which yields

$$z_j = tanh(a_j) \tag{26}$$

The output unit, z_k, is going to sum the linear combinations of z_j and transform it with its piecewise linear activation function described above. That is

$$z_{out} = \sum_M w_{kj} z_j \tag{27}$$

gets transformed by $g(\cdot)$ which yields

$$y = g(z_{out}) \tag{28}$$

where

$$g(z_{out}) = \begin{cases} 1 & z_{out} = M \\ 0 & o.w. \end{cases} \tag{29}$$

where M is the number of hidden units. Now, lets define the error function for each input vector, x^n, as the sum-of-squares function:

$$E^n = \frac{1}{2}(y - t)^2 \tag{30}$$

where t_k is the target value associated with a feature vector. The derivative of E^n w.r.t. the activation function, which is simply y_k, is given by

$$\delta_{out} = \frac{\partial E^n}{\partial a_{out}} = \frac{\partial E^n}{\partial y} = y - t \tag{31}$$

Now, for the hidden layers we use the definition of δ_j defined above. Note: $g'(a) = sech^2(a)$ since we are using the hyperbolic tangent activation function. Therefore, the derivative of E^n w.r.t. the activation function is

$$\delta_j = sech^2(\sum_i w_{ji}x_i)w_j\delta_{out} \tag{32}$$

Now, we have the derivative of the the error for each input vector w.r.t. each weight for each layer. The first layer is

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j x_i = sech^2(\sum_i w_{ji}x_i)w_j(y-t)x_i \tag{33}$$

and the second layer is

$$\frac{\partial E^n}{\partial w_{kj}} = \delta_{out} z_j = (y-t)tanh(\sum_i w_{ji}z_i) \tag{34}$$

Now we can use the gradient descent algorithm to update the weights. The recursive function for the first weights will be updated according to

$$w_{ji}^{\tau+1} = w_{ji}^{\tau} + \eta \cdot sech^2(\sum_i w_{ji}x_i)w_j(y-t)x_i \tag{35}$$

and the recursive function for the second layer of weights will update according to

$$w_{kj}^{\tau+1} = w_{kj}^{\tau} + \eta \cdot (y-t)tanh(\sum_i w_{ji}x_i) \tag{36}$$

By the perceptron convergence theorem, if the data is linearly separable by a finite union of hyperplanes, then a solution will be found in a finite number of steps. Convergence to local optima becomes a problem once the output activation function is not linear and continuous.

4. Learning and Generalization

The goal is to train the Neural Network in order to make a generalization about the underlying process which controls the behavior in the data. In order to properly generalize, one must take into consideration the bias and variance of the models. Bias represents how well the network can *learn* or *generalize* the underlying behavior. Variance usually reflects the complexity of the model. Typically, as the complexity of the model increases the bias decreases and the variance increases. The optimal point of generalization is the point at which the rate of the Bias is equal to the negative rate of the variance. This will provide the optimal level of complexity which fits the model. Overfitting will occur if we use a level of complexity (more hidden nodes) than this optimal point suggests. Underfitting will occur when we use less nodes than suggested by this optimal point.

In order to get a better idea of *bias* and *variance*, lets take a close look at an error function of the Neural Network model. First, we can derive an error function using the maximum liklihood function of the underlying distribution of the data as follows

$$L = \prod_{n}^{\infty} p(\mathbf{x}^n, t^n) = \prod_{i=1}^{\infty} p(t^n|\mathbf{x}^n)p(\mathbf{x}^n) \tag{37}$$

Equivalently, we can minimize the negative log of L which we can call our error function

$$E = -ln(L) = -\sum_{n} \log p(t^n|\mathbf{x}^n) - \sum_{n} \log p(\mathbf{x}^n) \tag{38}$$

However, the second term, $\sum \log p(\mathbf{x}^n)$, does not depend on the model parameters so we can ignore this term. Now, we can consider multiple possible outputs indexed by k. If we assume the target values t_k are independent we can express the underlying distribution as

$$p(\mathbf{t}|\mathbf{x}) = \prod_{k=1}^{c} p(t_k|\mathbf{x}) \tag{39}$$

Furthermore, if we assume the target variables, t_k, can be modeled using some deterministic function, $h(\mathbf{x})$, plus some Gaussian error, ϵ_k, then we have

$$t_k = h(\mathbf{x}) + \epsilon_k. \tag{40}$$

Assuming the Gaussian error has $\mu = 0$, we can solve for ϵ and find the probability of the target given the input vector as

$$p(t_k|\mathbf{x}) = \frac{1}{(2\pi\sigma^2)^{\frac{1}{2}}} \exp\left(\frac{-\{y_k(\mathbf{x};\mathbf{w} - t_k)\}^2}{2\sigma^2}\right) \tag{41}$$

Substituting equation (42) into (40) will give us

$$E = \frac{1}{2\sigma^2}\sum_{n=1}^{N}\sum_{k=1}^{c}\{y_k(\mathbf{x}^n;\mathbf{w}) - t_k\}^2 + Nc\log(\sigma) + \frac{Nc}{2}\log(2\pi). \tag{42}$$

Again, we can ignore the second and third terms which do not depend on the parameter. If we multiply (43) by $\frac{1}{N}$, and take the limit, then we will get a *Riemann-Stieltjes Integral* of E w.r.t the underlying density function, $p(t_k, \mathbf{x})$ as follows

$$\begin{aligned} E &= \lim_{N\to\infty} \frac{1}{2N}\sum_{n=1}^{N}\sum_{k}\{(y_k(\mathbf{x}^n;\mathbf{w}) - t_k^n\}^2 \\ &= \frac{1}{2}\sum_{k}\int\int\{y_k(\mathbf{x};\mathbf{w}) - t_k\}^2 p(t_k, \mathbf{x})\, dt_k\, d\mathbf{x}. \end{aligned} \tag{43}$$

By adding and subtracting the conditional expectation of the target given the input vector we can re-express the term in the brackets from (44) as

$$\begin{aligned} \{y_k - t_k\}^2 &= \{y_k - \mathbf{E}[t_k|\mathbf{x}] + \mathbf{E}[t_k|\mathbf{x}] - t_k\}^2 \\ &= \{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2 + 2\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}\{\mathbf{E}[t_k|\mathbf{x}] - t_k\} \\ &+ \{\mathbf{E}[t_k|\mathbf{x}] - t_k\}^2 \end{aligned} \tag{44}$$

If we substitute (45) into (44) and factor the joint density into the density of the target conditioned on the input vector and the unconditional density of the input vector, then we will get

$$\frac{1}{2}\sum_k \int\int \Big[\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2 \;+\; 2\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}\{\mathbf{E}[t_k|\mathbf{x}] - t_k\} \\ +\; \{\mathbf{E}[t_k|\mathbf{x}] - t_k\}^2\Big] p(t_k|\mathbf{x})p(\mathbf{x})\, dt_k\, d\mathbf{x}$$

First, lets evaluate the inner integral w.r.t dt_k.

$$\int \Big[\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2 + 2\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}\{\mathbf{E}[t_k|\mathbf{x}] - t_k\} + \{\mathbf{E}[t_k|\mathbf{x}] - t_k\}^2\Big] p(t_k|\mathbf{x}) dt_k$$

$$= \{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2 + 2\{y_k - \mathbf{E}[t_k|\mathbf{x}]\} \int \{\mathbf{E}[t_k|\mathbf{x}] - t_k\} p(t_k|\mathbf{x}) dt_k + \\ \int \Big[\mathbf{E}[t_k|\mathbf{x}]^2 - 2t_k\mathbf{E}[t_k|\mathbf{x}] + t_k^2\Big] p(t_k|\mathbf{x}) dt_k$$

$$= \{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2 + 2\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}\{\mathbf{E}[t_k|\mathbf{x}] - \mathbf{E}[t_k|\mathbf{x}]\} + \mathbf{E}[t_k|\mathbf{x}]^2 - 2\mathbf{E}[t_k|\mathbf{x}]^2 + \mathbf{E}[t_k^2|\mathbf{x}]$$

$$= \{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2 + \mathbf{E}[t_k^2|\mathbf{x}] - \mathbf{E}[t_k|\mathbf{x}]^2 \tag{45}$$

In order to see this through its helpful to recall $\mathbf{E}[t_k|\mathbf{x}] = \int t_k p(t_k|\mathbf{x}) dt_k$, $\mathbf{E}[t_k^2|\mathbf{x}] = \int t_k^2 p(t_k|\mathbf{x}) dt_k$, and note that $\mathbf{E}[t_k|\mathbf{x}]$ and y_k are not functions of t_k. Since we are only concerned with the weight parameters $\mathbf{w}$, the error function to minimize can be expressed as

$$\frac{1}{2}\sum_k \int \{y_k(\mathbf{x};\mathbf{w}) - \mathbf{E}[t_k|\mathbf{x}]\}^2 p(t_k|\mathbf{x})p(\mathbf{x})\, dt_k\, d\mathbf{x}. \tag{46}$$

The minimum is clearly $y_k(\mathbf{x};\mathbf{w}) = \mathbf{E}[t_k|\mathbf{x}]$ which expresses the idea that the $\mathbf{w}*$ which does minimize this error will create a output y_k which is the average of t_k for a given $\mathbf{x}$. In other words, the network model is given by the regression of t_k conditioned on $\mathbf{x}$.

4.1. Bias and Variance

In order to express the error as a sum of the bias and variance, we need to find the ensemble average of the $\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2$ term. The ensemble average, $\mathbf{E}_D[\cdot]$, will express the nature of $\{y_k - \mathbf{E}[t_k|\mathbf{x}]\}^2$ independent of the training set by considering an "ensemble" of data sets. This allows us to take a look at how the model with weights, $\mathbf{w}$, will generalize the behavior of the underlying distribution. First, we look at the term in (47) and add and subtract the ensemble average of the model and expand and find the ensemble average as follows

$$\{y - \mathbf{E}[t|\mathbf{x}]\}^2 = \{y - \mathbf{E}_D[y(\mathbf{x})] + \mathbf{E}_D[y(\mathbf{x})] - \mathbf{E}[t|\mathbf{x}]\}^2$$

$$= \{y(\mathbf{x}) - \mathbf{E}_D[y(\mathbf{x})]\}^2 + \{\mathbf{E}_D[y(\mathbf{x})] - \mathbf{E}[t|\mathbf{x}]\}^2 + 2\{y(\mathbf{x}) - \mathbf{E}_D[y(\mathbf{x})]\}\{\mathbf{E}_D[y(\mathbf{x})] - \mathbf{E}[t|\mathbf{x}]\} \tag{47}$$

Now, if we take the ensemble expectation over (48) and also note the only parameter which is not constant is $y(\mathbf{x})$, then we get

$$\mathbf{E}_D[\{y - \mathbf{E}[t|\mathbf{x}]\}^2]$$

$$= \{\mathbf{E}_D[y(\mathbf{x})] - \mathbf{E}[t|\mathbf{x}]\}^2 + \mathbf{E}_D[\{y(\mathbf{x}) - \mathbf{E}_D[y(\mathbf{x})]\}^2] \tag{48}$$

The first term is referred to as the bias squared and the second term is the variance. We can find the average bias and variance by integrating over the density of $\mathbf{x}$. That is,

$$(\text{Bias})^2 = \frac{1}{2}\int \{\mathbf{E}_D[y(\mathbf{x})] - \mathbf{E}[t|\mathbf{x}]\}^2 p(\mathbf{x}) d\mathbf{x} \tag{49}$$

$$\text{Variance} = \frac{1}{2}\int \mathbf{E}_D[\{y(\mathbf{x}) - \mathbf{E}_D[y(\mathbf{x})]\}^2] p(\mathbf{x}) d\mathbf{x} \tag{50}$$

5. Example Data and Matlab Implementation

This section explores a few different data sets and trains a network accordingly. The Matlab code for various network structures is in the Section Matlab Code. We will designate a subsection to each data set and discuss the network parameters and topology. Each network is trained using the back-propagation method which implements the gradient descent. The activation function used is the sigmoid function. There are many ways to implement the back-propagation routine much faster [7], [11], [3]. Furthermore, it is important to keep in mind the rate of convergence. There can be many local optima in the error if there are at least one hidden unit. As the complexity increases oscillations between local minimum errors seem to occur.

Certain data sets will suggest different network structures. There is an *ad hoc* method to "prune" the network according to the data set. If there are many weights close to zero in any of the weight matrices, then we can reduce the number of nodes in the layer operated on by the weights. So, often a structure with many nodes in each hidden layer will be initialized and the number of nodes will be reduced after each training. This also ensures we do not overfit the data.

5.1. Simple Linear Classification

In Matlab, points were sampled from the unit square. All points above $y = x$ will be considered in class 1 and all points under $y = x$ will be in class 2. The data is plotted in Figure 3.

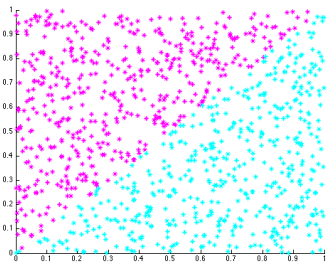

Figure 3: Plot of data

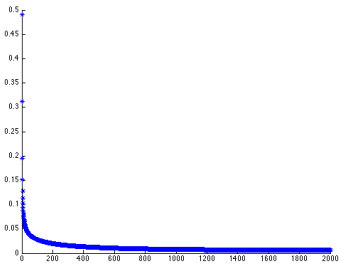

Figure 4: Error converges to zero

Although a network with no hidden layer can learn this training set, a network with a hidden layer is used. There are 2 input nodes, 2 hidden nodes, and one output node. The rate of convergence for the gradient descent was set to $\eta = 2.1$. The error associated with this model converged to zero. Figure 4 displays a plot of the networks error as the number of epochs increases.

5.2. Nonlinear Classification

Points were sampled from the unit square. All points above $y = .5 * cos(.3 * x) * cos(10 * x) + .5$ were classified as Class 1 and all points below as Class 2. The data is plotted in Figure 5.

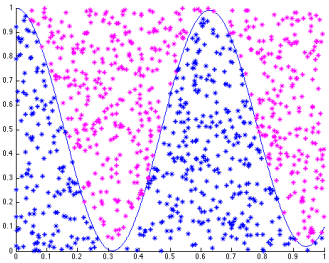

Figure 5: Plot of data

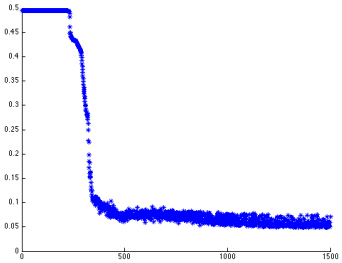

Figure 6: Error vs. number of epochs

A network with 2 input nodes, 10 nodes in the first hidden layer, 20 nodes in the second, 5 the the third and 1 output is trained. The learning rate for the gradient descent algorithm is $\eta = .6$ The network structure is fairly complex for a reason. A better representation is needed for this data. Recall our activation function is the sigmoid function. This function reaches 0 or 1 in the limit. It would be appropriate the scale the decision boundary or data so that the upper and lower bound lies well within [0,1]. Nonetheless, the network error reaches above 90% accuracy. It is also important to node that the density of points is uniform throughout the unit square. In reality, the density of the points near a decision boundary is less. This also effects the rate of convergence.

5.3. Nested Data

Points in Class 1 were sampled from a disk inside points of Class 2 that constructs a washer. Ultimately, a representation of the data needs to be achieved by the composition of linear transformations from the weight matrices and point wise activation function in order to linearly separate the data. The data is plotted in Figure 7.

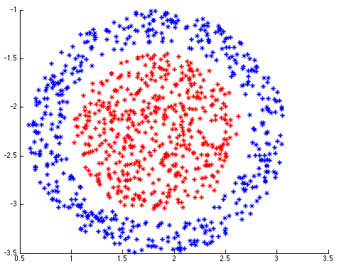

Figure 7: Plot of data

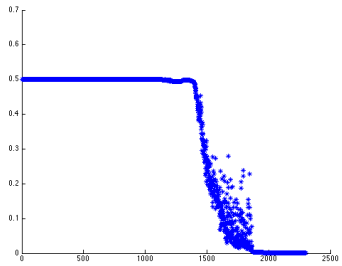

Figure 8: Error vs. number of epochs

A network with 2 input nodes, 14 nodes in first hidden layer, 7 in the second hidden layer, 4 nodes in the third hidden layer and 1 output layer was trained. Particular attention to the learning rate is crucial since the high level of complexity leads to many local optima in the error. The learning rate is set to $\eta = .3$. This network structure is capable of learning the data set and error converges to zero.

5.4. Multiple Classes

A simple data set was constructed to partition the unit square into quadrants for four classes. The data is plotted in Figure 9. The network has 2 input nodes, 10 nodes in the

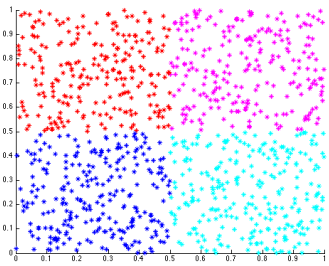

Figure 9: Plot of data

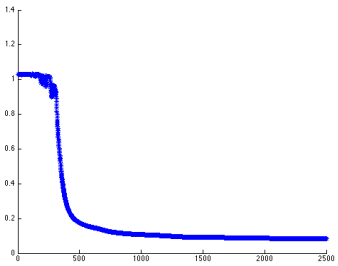

Figure 10: Error vs. number of epochs

first hidden layer, 10 nodes in the second hidden layer, and 1 output. The learning rate is $\eta = .9$ Again, the uniform density close to the boundary leads to higher error. The error converges, but in the limit of the number of epochs because of uniform density which is simply a result of the pseudo-data.

6. Conclusion

This paper simply explored the surface and fundamentals of Artificial Neural Networks. It is important to keep in mind that they are often used in a Machine Learning setting. I provided no examples of model validation from the training and testing perspective. I simply explored the basis of neural networks, defined the network structure and parameters, considered the bias and variance dilemma, and trained a few networks for different data set types. Overall, the Artificial Neural Network seems to be robust to various types of data sets. They are also relatively intuitive to understand. Studying them in order from the single perceptron and non-hidden layer networks to more complex networks provides a solid framework to understand the basis of more complex structures.

There are many ways of implementing the neural network. The convolution network is often used for image processing [4], the recurrent neural network which creates a directed cycle between hidden nodes and can capture dynamic temporal behavior, time delay neural networks which are used for sequential data [12], non-parametric kernel density estimator neural networks (Probability Neural Networks) [9], among many other types.

An important use of Neural Networks is regression. Although, this paper did not explicitly discuss regression, the idea is very similar. Instead of the output activation function being sigmoidal, it is simply the identity function. In this setting, the neural network can create a general nonlinear regression model. Overall, the neural network and many other machine learning algorithms should be introduced at the undergraduate level. They are robust, their framework extends to various applications, and coding them provides a much better understanding of the mechanics and reason why statistical models work.

7. Matlab Code

7.1. One Hidden Layer Network

```
1
2  function [w] = neural(Attributes, Classifications)
3
4  n = .3;
5  nbrOfNodes = 2;
6  nbrOfEpochs = 2000;
7
8  % Initialize matrices with random weights 0-1
9  W = rand(nbrOfNodes, length(Attributes(1,:)));
10 U = rand(length(Classifications(1,:)),nbrOfNodes);
11
12 m = 0; figure; hold on; e = size(Attributes);
13
14 while m < nbrOfEpochs
15
16     % Increment loop counter
17     m = m + 1;
18
19     % Iterate through all examples
20     for i=1:e(1)
21         % Input data from current example set
22         I = Attributes(i,:).';
23         D = Classifications(i,:).';
24
25         % Propagate the signals through network
26         H = f(W*I);
27         O = f(U*H);
28
29         % Output layer error
30         delta_i = O.*(1-O).*(D-O);
31
32         % Calculate error for each node in layer_(n-1)
33         delta_j = H.*(1-H).*(U.'*delta_i);
34
35         % Adjust weights in matrices sequentially
36         U = U + n.*delta_i*(H.');
37         W = W + n.*delta_j*(I.');
38     end
39
40     RMS_Err = 0;
41
42     % Calculate RMS error
43     for i=1:e(1)
44         D = Classifications(i,:).';
```

```
45              I = Attributes(i,:).';
46              RMS_Err = RMS_Err + norm(D-f(U*f(W*I)),2);
47
48          end
49
50          y = RMS_Err/e(1);
51          plot(m,y,'*');
52      w=W;
53      end
54
55
56      function x = f(x)
57      x = 1./(1+exp(-x));
```

7.2. Two Hidden Layer Network

```
function [w1,w2,w3] = neuralNet1(Attributes,
    Classifications)

n = .2;
nbrOfNodes1 = 15;
nbrOfNodes2 = 3;
nbrOfEpochs = 2500;

% Initialize matrices with random weights 0-1
W1 = rand(nbrOfNodes1, length(Attributes(1,:)));
W2 = rand(nbrOfNodes2,nbrOfNodes1);
W3 = rand(length(Classifications(1,:)),nbrOfNodes2);

m = 0; figure; hold on; e = size(Attributes);

while m < nbrOfEpochs

    % Increment loop counter
    m = m + 1;

    % Iterate through all examples
    for i=1:e(1)
        % Input data from current example set
        I = Attributes(i,:).';
        D = Classifications(i,:).';

        % Propagate the signals through network
        H1 = f(W1*I);
        H2 = f(W2*H1);
        O = f(W3*H2);

        % Output layer error
        delta_i = O.*(1-O).*(D-O);

        % Calculate error for each node in layer_(n-1)
        delta_j = H2.*(1-H2).*(W3.'*delta_i);
        delta_k = H1.*(1-H1).*(W2.'*delta_j);

        % Adjust weights in matrices sequentially
        W3 = W3 + n.*delta_i*(H2.');
        W2 = W2 + n.*delta_j*(H1.');
        W1 = W1 + n.*delta_k*(I.');
    end

    RMS_Err = 0;
    % Calculate RMS error
```

```
46      for i=1:e(1)
47          D = Classifications(i,:).';
48          I = Attributes(i,:).';
49          RMS_Err = RMS_Err + norm(D-f(W3*f(W2*f(W1*I))),2);
50
51      end
52      y = RMS_Err/e(1);
53      plot(m,y,'*');
54
55      w1=W1;
56      w2=W2;
57      w3=W3;
58  end
59
60
61  function x = f(x)
62  x = 1./(1+exp(-x));
```

7.3. Three Hidden Layer Network

```
function [w1, w2, w3, w4] = neuralNet2(Attributes,
    Classifications)

n = .3;
nbrOfNodes1 = 10;
nbrOfNodes2 = 20;
nbrOfNodes3 = 5;
nbrOfEpochs = 1500;

% Initialize matrices with random weights 0-1
W1 = rand(nbrOfNodes1, length(Attributes(1,:)));
W2 = rand(nbrOfNodes2,nbrOfNodes1);
W3 = rand(nbrOfNodes3,nbrOfNodes2);
W4 = rand(length(Classifications(1,:)),nbrOfNodes3);
m = 0; figure; hold on; e = size(Attributes);

while m < nbrOfEpochs

    % Increment loop counter
    m = m + 1;

    % Iterate through all examples
    for i=1:e(1)
        % Input data from current example set
        I = Attributes(i,:).';
        D = Classifications(i,:).';

        % Propagate the signals through network
        H1 = f(W1*I);
        H2 = f(W2*H1);
        H3 = f(W3*H2);
        O = f(W4*H3);

        % Output layer error
        delta_i = O.*(1-O).*(D-O);

        % Calculate error for each node in layer_(n-1)
        delta_j = H3.*(1-H3).*(W4.'*delta_i);
        delta_k = H2.*(1-H2).*(W3.'*delta_j);
        delta_l = H1.*(1-H1).*(W2.'*delta_k);

        % Adjust weights in matrices sequentially
        W4 = W4 + n.*delta_i*(H3.');
        W3 = W3 + n.*delta_j*(H2.');
        W2 = W2 + n.*delta_k*(H1.');
        W1 = W1 + n.*delta_l*(I.');
```

```
46      end
47
48      RMS_Err = 0;
49
50      % Calculate RMS error
51      for i=1:e(1)
52          D = Classifications(i,:).';
53          I = Attributes(i,:).';
54          RMS_Err = RMS_Err + norm(D-f(W4*f(W3*f(W2*f(W1*I))))
                ,2);
55
56      end
57
58      y = RMS_Err/e(1);
59
60      plot(m,y,'*');
61      w1=W1;
62      w2=W2;
63      w3=W3;
64      w4=W4;
65
66  end
67
68
69  function x = f(x)
70  x = 1./(1+exp(-x));
```

References

[1] Christopher M Bishop et al. Neural networks for pattern recognition. 1995.

[2] HD Block. The perceptron: A model for brain functioning. i. *Reviews of Modern Physics*, 34(1):123, 1962.

[3] Scott E Fahlman. An empirical study of learning speed in back-propagation networks. 1988.

[4] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[5] John Hertz. *Introduction to the theory of neural computation*, volume 1. Basic Books, 1991.

[6] Richard P Lippmann. An introduction to computing with neural nets. *ASSP Magazine, IEEE*, 4(2):4–22, 1987.

[7] Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

[8] Terence D Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural networks*, 2(6):459–473, 1989.

[9] Donald F Specht. Probabilistic neural networks. *Neural networks*, 3(1):109–118, 1990.

[10] Kenneth O Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. *Network (Phenotype)*, 1(2):3, 1996.

[11] Thomas P Vogl, JK Mangis, AK Rigler, WT Zink, and DL Alkon. Accelerating the convergence of the back-propagation method. *Biological cybernetics*, 59(4-5):257–263, 1988.

[12] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(3):328–339, 1989.