Teaching a Computer About Balance: An Neural Network Exploration

Erik Shimshock
Scientific Computing
Harvey Mudd College

May 5, 2006

1 Motivation

Though I've decided to focus on artificial intelligence in graduate school next year, I've never had a chance to take a course in neural networks or artificial intelligence. This project seemed like a great opportunity for me to explore some of what I missed out on. Not knowing where to start, Professor Keller who periodically teaches Neural Networks, kindly pointed me towards multilayer perceptrons trained using back propagation.

2 Introduction

A neural network is a computational model inspired by systems of connected biological neurons. The type of neural network in this project is referred to as a *multilayer perceptron*. These consist of multiple layers of nodes, the first being the *input layer*, the last being the *output layer*, and all those in between are the *hidden layers*. Nodes are modeled after biological neurons; the output of a specific node is obtained by summing up the incoming signals and apply a nonlinear activation function. A computation using the neural network involves inputting numerical values into the first layer and observing the values output at the last layer. Signals are propagated from layer to layer through weighted connections between all of the nodes in one layer and all of the nodes in the next layer. Figure 1 depicts a simple multilayer perceptron which receives two inputs, produces one output value, and contains 1 hidden layer with two nodes in it.

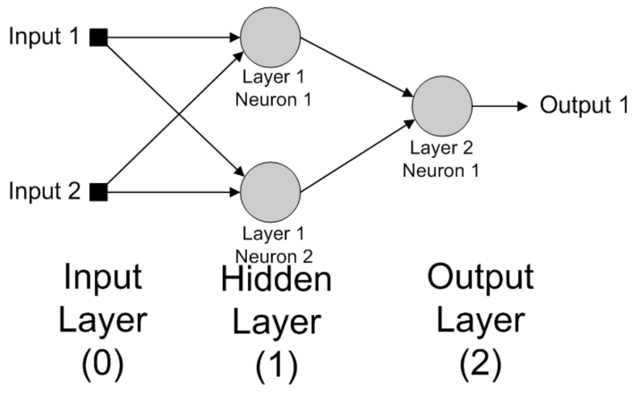

Figure 1: A simple multilayer perceptron

3 Model Details

3.1 Forward Computation

Consider a multilayer perceptron which has m nodes in its input layer. Let x_1, $x_2, \ldots, x_m$ be a specific set of inputs these nodes. The net input to node j in layer 1 (the first hidden layer), $v_j^{(1)}$, is

$$v_j^{(1)} = \sum_{i=0}^{m} w_{ji}^{(1)} x_i \tag{1}$$

where $w_{ji}^{(1)}$ is a real number representing the strength of the connection from input node i. Notice that the index i actually starts at 0; so what are $w_{j0}^{(1)}$ and x_0? The product $w_{j0}^{(1)} x_0$ describes the default behavior of the node when all of the real inputs sum to zero. The value of x_0 is defined to be -1, so $w_{j0}^{(1)}$ defines what is called the *threshold* of the node (it is sometimes distinguished as $\theta_j^{(1)}$ but I find it easier to think of it as just another connection weight value, but one which is always connected to an output of -1). In general the net input to node j in layer

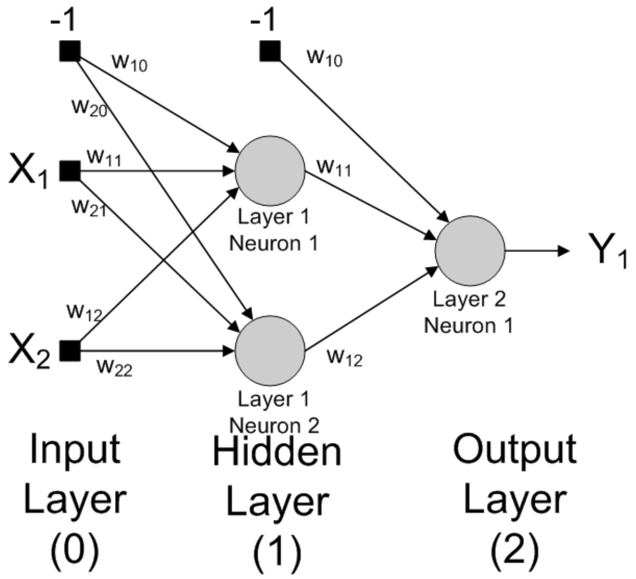

Figure 2: A more detailed look at a simple multilayer perceptron

l, $v_j^{(l)}$, is

$$v_j^{(l)} = \sum_{i=0}^{p} w_{ji}^{(l)} y_i^{(l-1)} \tag{2}$$

where p is the number of nodes in the previous layer (layer $l-1$), $w_{ji}^{(l)}$ is the strength of the connection from node i in the previous level, and $y_i^{(l-1)}$ is the output of node i in the previous level. Just as before, $y_i^{(l-1)}$ is defined to be -1 and $w_{j0}^{(l)}$ is threshold for node j. To get a clearer idea of how these all of these parameters interact, take a look at Figure 2 which shows a more detailed look at the example network.

The output of node j in layer l is, as briefly mentioned earlier, a nonlinear function of the net input to node j, specifically

$$y_j^{(l)} = \phi(v_j^{(l)}). \tag{3}$$

Two popular choices for the function ϕ are the logistic function and the hyperbolic tangent,

$$y_j = \phi(v_j) = \frac{1}{1+e^{-v_j}} \qquad \text{(logistic function)}, \tag{4a}$$

$$y_j = \phi(v_j) = \frac{2}{1+e^{-v_j}} - 1 \qquad \text{(hyperbolic tangent)}, \tag{4b}$$

whose behavior is shown in Figure 3. Note that the logistic function asymptotically approaches the values 1 and 0 as the input tends towards ∞ and $-\infty$ respectively. Similarly the hyperbolic tangent function approaches 1 and -1 as the input tends towards ∞ and $-\infty$.

Starting with the input values and iteratively applying these rules it is straightforward to compute the output values of the neural network.

3.2 Training

Sure, computing the output of a specific neural network is easy, but so far it isn't very interesting. We don't want the neural network to produce some arbitrary output, we want it to learn something. Specifically, we want to present it with a set of training data, have it learn from this, and hopefully it will be able to generalize beyond the examples it was trained on. A training example is a known input-output pair, $[\mathbf{x},\mathbf{d}]$, where $\mathbf{x}$ is the vector of values for the input nodes and $\mathbf{d}$ is the vector of values the output nodes should produce.

One method for teaching a neural network is the *back-propagation* algorithm. The idea is to run an example input through the network and determine how far off its output is from the desired output. Then using the errors at the output nodes we estimate the errors at the previous level's nodes. At the same time we get an estimate at how much the error is due to each connection from the previous layer and adjust those connection weights accordingly. We repeat this process, using the estimates for the node errors at a specific layer to estimate the previous layer's node errors and to also adjust the connection weights, until we reach the input layer.

The error estimates begin with the calculation of the known output layer errors. Assume the network has L layers, and thus for a specific training example the error at node j in the output layer L, $e_j^{(L)}$, is

$$e_j^{(L)} = d_j - o_j \tag{5}$$

where d_j is the desired output for the node and and o_j is the actual output of the node. We estimate the local gradient of the error for node j in the output layer, $\delta_j^{(L)}$, as

$$\delta_j^{(L)} = \phi'(o_j)e_j^{(L)} \tag{6}$$

where ϕ' is the derivative of the chosen neural activation function (see equation (4)). For a node in a hidden layer l, the local gradient for node j, $\delta_j^{(l)}$, is

$$\delta_j^{(l)} = \phi'(y_j^{(l)}) \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \tag{7}$$

recalling that $y_j^{(l)}$ is the output of node j in layer l and $w_{kj}^{(l+1)}$ is the weight of the connection from node j in layer l to node k in layer $l+1$.

We use these local gradient estimates to determine the adjustments we'll make to the connection weights. The simplest rule for determining the adjustment to the connection from node i in layer $l-1$ to node j in layer l, $\Delta w_{ji}^{(l)}$, is

$$\Delta w_{ji}^{(l)} = \eta \delta_j^{(l)} y_i^{(l-1)} \tag{8}$$

where η is the *learning-rate* parameter, a constant between 0 and 1. Low values of η result in slow learning, but high values of η can result in unstable oscillating weight changes. One common modification involves adding a *momentum* term,

$$\Delta w_{ji}^{(l)} = \alpha \cdot \Delta' w_{ji}^{(l)} + \eta \delta_j^{(l)} y_i^{(l-1)}, \tag{9}$$

where $\Delta' w_{ji}^{(l)}$ is the previous change applied to this connection weight and α is the *momentum constant* with a value between 0 and 1. This modification can allow quicker convergence while maintaining stability.

Using a specific training point, this method allows us to determine the modifications for all of the weights, level-by-level working back-to-front. Running through the rest of the of the training examples is referred to as an *epoch*, and we repeat epoch after epoch until the network has satisfactorily learned the training set. At this point the network is now trained and can hopefully accurately produce outputs for the known inputs and generalize to inputs it didn't train on.

4 Implementation

I implemented this algorithm in MATLAB. In planning out how the various variables would be stored, I came to a profound (to me at least) realization. It made sense to keep track of the various values in matrices and vectors (e.g. the weights from from layer 0 to layer 1 would be a matrix, the gradients for the nodes in layer 1 would be a vector, etc.), but if I stored them in the right ways I could cleverly not just some, but *ALL* of the calculations in the algorithm using algebraic matrix operations. For example with the neural network in Figure 2, equation (1) for calculating the net inputs to the layer 1 nodes can be formulated as

$$\begin{bmatrix} v_1^{(1)} & v_2^{(1)} \end{bmatrix} = \begin{bmatrix} -1 & x_1 & x_2 \end{bmatrix} \cdot \begin{bmatrix} w_{10} & w_{20} \\ w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix}.$$

In this way I was able to deal calculate the a set of variables for a layer in one fell swoop using matrix additions, multiplications and the occasional transpose. One awesome side effect of coding the algorithm in this way was that I was able to entirely avoid the annoyance of MATLAB indexing its matrix and vectors starting at 1.

5 Application

At this point I was like a fisherman in the desert; I had a neural network but no data set to try it on. I could think of lots of interesting things to try and get my network to learn, but none of them had a large set of known input-output pairs to use as a training set.

5.1 Scale Balancing

After some searching, I eventually found an interesting problem and data set on the internet. Coincidentally the data set was provided by the graduate school program I had visited a couple of weeks earlier. From the many data sets on the UC Irvine Machine Learning Repository I chose the Balance Scale Database [D.J. Newman and Merz(1998)]. The problem set before the neural network was to determine if a scale was left-heavy, balanced, or right-heavy given the masses and distances from the center for a weight on the left and right of the scale. The data set had 5 possible weight and distance values, totaling $625 = 5^4$ data points.

5.1.1 Methodology

My neural network consisted of 4 input nodes, 1 output node, and a single hidden layer with 2 nodes. For a left-leaning scale the desired output would be -1, for a balanced scale the desired output would be 0, and for a right-leaning scale the desired output would be 1. I used a learning-rate parameter, $\eta = .01$, and momentum constant, $\alpha = 0.9$. My training set consisted of a random 50% of the data set. I ran a training session of 500 epochs, and in each epoch the order that the training examples were presented to the network was randomized.

5.1.2 Results

The standard measure of performance for a specific run of a neural network is the *instantaneous sum of squared errors* defined as

$$\mathcal{E} = \frac{1}{2}\sum_j e_j^2 \tag{10}$$

where the sum is over all output nodes, and e_j is the error for the jth output node. In my case there is only one output neuron so the value is merely half of the error squared. Measuring the *average squared error* is accomplished by averaging over a whole epoch. At the end of 500 training epochs, testing on the training set resulted in a good average squared error of 0.0332. Testing this trained network on the whole data set resulted in an average squared error of 0.0712, showing that clearly the network *was* able to take what it had learned and generalize to most of the rest of the data set. In fact, for 90% of the data points, the network achieved an instantaneous squared error of less than 0.125. This means that for 90% of the inputs, rounding the output of the network to the nearest integer results in the correct answer. Figure 4 shows the distribution of errors across the whole data set.

Looking more closely at which data points were having the larger errors, I noticed a pattern. I realized that the larger errors seemed to occur when the scale was almost balanced (the converse was not always true since there were many closely balanced points which it correctly predicted). I was able to quantify this observation by calculating the difference magnitude,

$$m_r d_r - m_l d_l$$

where w_r and w_l are the masses and d_r and d_l are the distances from the center for the right and left weights (so if the value is negative the scale is tipped to the left, 0 its balanced, and positive its tipped to the right). Figure 5 shows a scatter plot of the difference magnitude versus the instantaneous squared error for each point in the data set, clearly confirming the pattern I noticed. Thinking about this, I realized that this is the region where small changes in the inputs result in large changes in the outputs (i.e. small changes to the masses or distances can change the balance of the scale drastically), so it is quite expected that the network would have a hard time making decisions there.

5.2 XOR Gate

As a proof of concept when I was initially coding up the neural network I used the simple `XOR` Gate data set. This data set has two binary input values and 1 binary output which is the exclusive-or of the inputs (1 if the inputs are different and 0 if the inputs are the same). With only 4 data points, this data set was clearly not for teaching a network to generalize. My network consisted of two input nodes, 1 output node, and 2 hidden nodes, just like the network in Figure 2. I used all 4 data points as my training set, and used it to verify that my neural network would in fact improve with training. I also used the data set to explore how changing the learning constant η and momentum constant α affected convergence speed and stability. Figures 6, 7, 8, and 9 plot the average squared error over each epoch for various values of the two parameters. I was able to verify the observation that combinations of a low learning rate and high momentum or high learning rate and low momentum were stable and converged.

References

[D.J. Newman and Merz(1998)] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998. URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[Haykin(1994)] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan College Publishing Company, 1994.

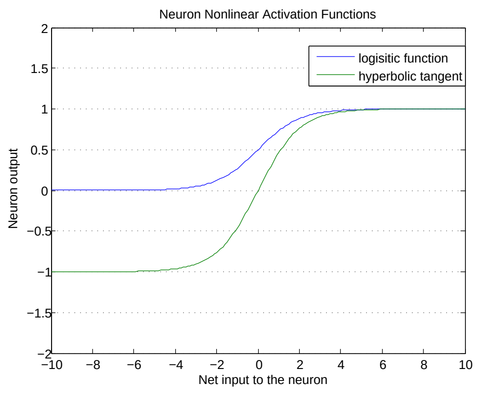

Figure 3: Two common choices for the node's activation function

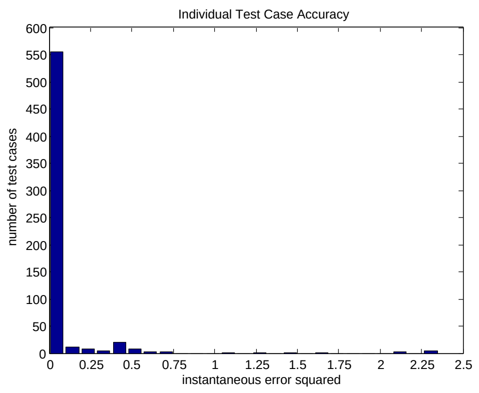

Figure 4: This shows the distribution of error magnitudes for the whole data set.

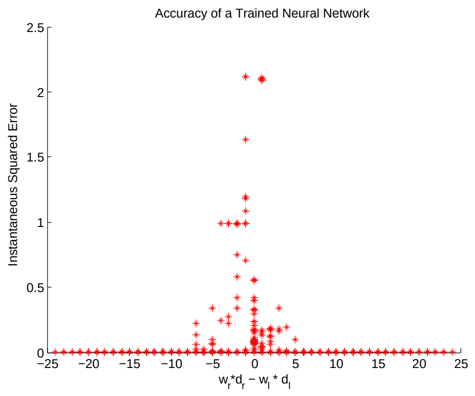

Figure 5: This shows that the points where the network had a hard time correlated with being nearly balanced.

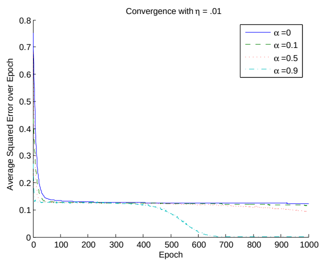

Figure 6: Varying momentum with constant learning value

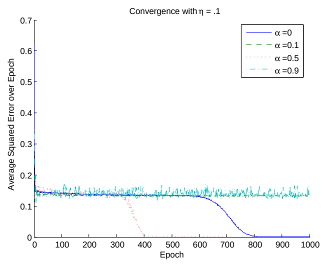

Figure 7: Varying momentum with constant learning value

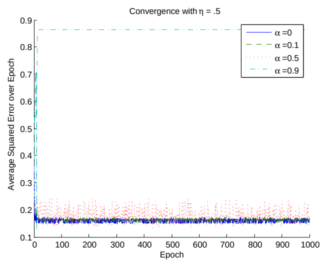

Figure 8: Varying momentum with constant learning value

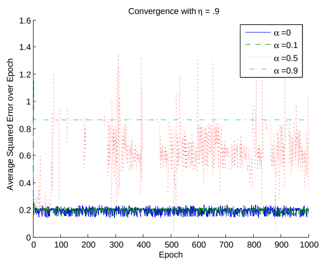

Figure 9: Varying momentum with constant learning value