Machine Learning
Neural Network Implementation

Adam Holmes
June 1 2017

1 Design Decisions

1.1 Structure

The general structure of my implementation attempts to clearly expose which components of the description of the feed-forward neural network technique are being implemented and where. Here I describe the major functions of the implementation:

- Preprocessing
- Math Helper Functions
- Function: Stochastic Gradient Descent
- Function: BackPropagation
- Function: UpdateWeights

1.1.1 Preprocessing

Program main begins by loading in training data by using pythons *loadtxt* functionality. The data is loaded into two separate multi-dimensional lists, one containing the 50,000 rows each comprised of 784 columns (one element per row per pixel of each image), and another list containing the appropriate data labels corresponding to each element. Additionally, support was added for argument parsing, to enable convenient data-sweeps across different learning rates, numbers of epochs, and hidden layer size.

Next, two sets of weights are constructed, one corresponding to the edges between the input set of neurons to the hidden layer, and the other corresponding to the edges between the hidden layer and the output layer. These weights are initialized randomly, sampling from a uniform normal distribution.

1.1.2 Math Helper Functions

Functions were designed to implement the sigmoid function and the derivative of the sigmoid function, respectively:

$$S(\alpha) = \frac{1}{1 + e^{\alpha}} \tag{1}$$

$$S'(\alpha) = S(\alpha)(1 - S(\alpha)) \tag{2}$$

Additionally, an *evaluate* function was added to enable quick and easy testing of new input values through the network.

1.1.3 Function: Stochastic Gradient Descent

This function is the highest-level function controlling the internal mechanics of the neural network. The function loops through a specified number of epochs, and for each epoch it iterates through each example and label pair in the training data. For each example, a new ∇_w is calculated utilizing the back propagation algorithm described in class, which is then fed into the weight updating function to actively update the weights. The granularity of the algorithm is that of individual examples: that is, weights are updated after each individual example. In this sense, no batching is performed, and the stochastic gradient descent takes a step towards the minimal value after processing each example.

Algorithm 1 Stochastic Gradient Descent Algorithm

for each epoch:
  for each example and label:
    $\nabla_w \leftarrow backPropagation(example, label, \eta, weights)$
    $weights \leftarrow updateWeights(\eta, \nabla_w, weights)$
  endfor
endfor

1.1.4 Function: BackPropagation

This algorithm implements both feed forward and back propagation, as described in class. Specifically, it creates two lists: activateList and intermediateList, which are responsible for containing the appropriate neuronal activations and intermediate dot products as necessary, respectively. Next, it calculates two different δ values, one used to update the weights set between the hidden layer and the output layer, and one used to update the remaining weights set. This is described more formally here.

These δ values are then used within outer products to construct matrices of appropriate dimension so as to update the weight matrices correctly.

Algorithm 2 Back Propagation Algorithm

$activateList \leftarrow [\boldsymbol{x}]$
$intermediateList \leftarrow [\varnothing]$
for w in $Weights$:
  $inter \leftarrow w \cdot activateList[-1]$
  $activate \leftarrow S(inter)$
  Push $inter$ to activateList, and $activate$ to intermediateList
$\delta_1 = activateList[2] - y_i \cdot S'(intermediateList[1])$
$\delta_2 = (weights[-1] \cdot \delta_1) \cdot S'(intermediateList[-2])$

1.1.5 Function: UpdateWeights

This function is a straightforward one that implements a modularized version of updating weights in accordance with the back propagation weight update rule.

2 Results and Discussion

The first presented graph is a sweep over a specific set of configurations, namely a hidden layer of size $N = 32$ neurons. The swept values are those of η, from 0.001 to 10.

As the plot clearly demonstrates, at this hidden layer size, an η value between 0.1 and 1 requires the fewest iterations of stochastic gradient descent to reach a minimal value. Interestingly, the asymptotic behavior is asymmetric: $\eta << 1$ shows slow convergence to minimal value with gradient descent, while $\eta >> 1$ appears to get stuck in a local minima that is very far from the other minima present.

A very similar trend is seen as the hidden layer size is increased to $N = 64$ neurons. These sweeps were kept to 16 epochs for time convenience.

Holding instead the η value constant (optimized value of 1.0), and sweeping instead over the size of the hidden layers, we find that the overall computation time increases substantially, while the overall performance picture remains somewhat constant.

The graphs accompany this document, on the following pages.

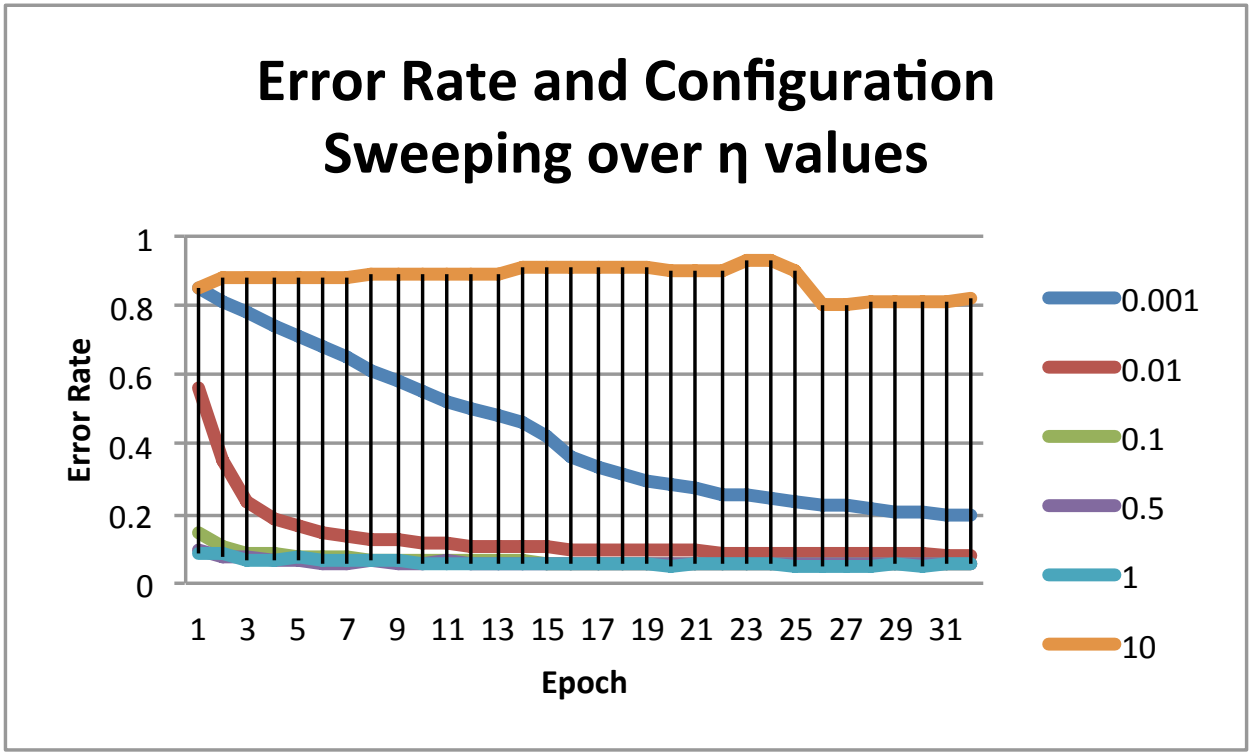

Figure 1: N = 32 Hidden Neurons

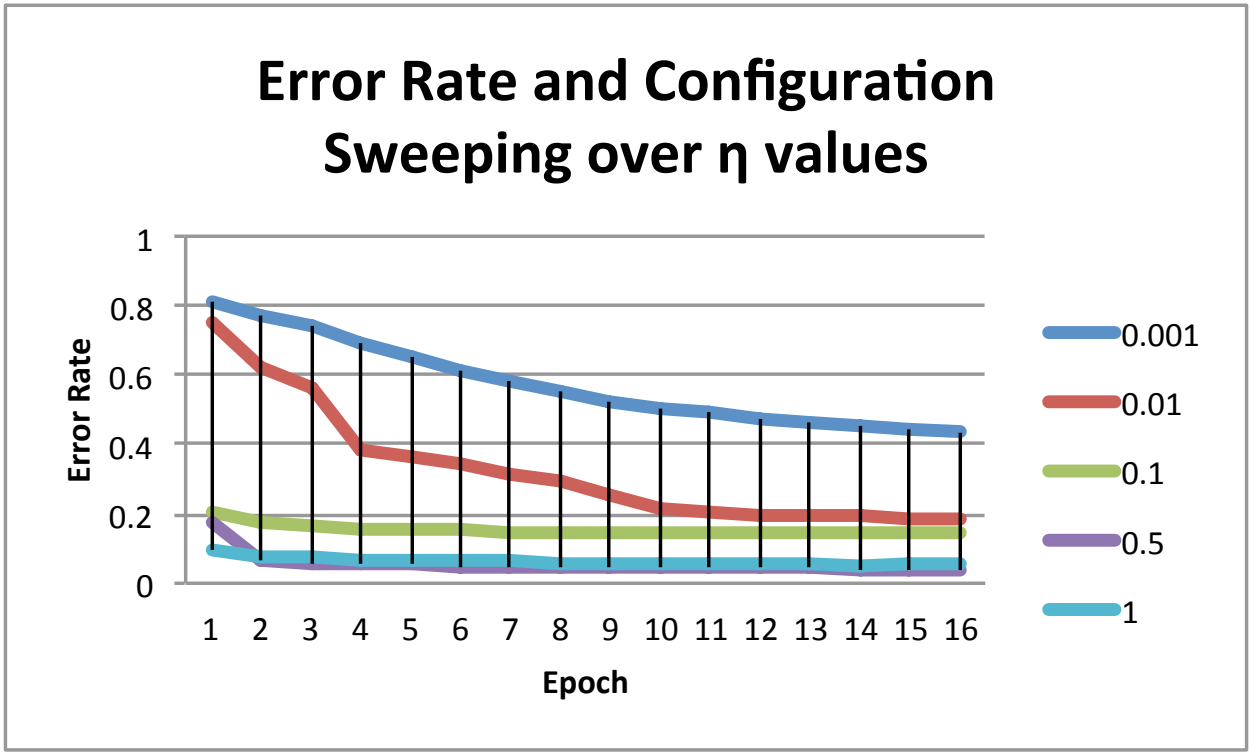

Figure 2: N = 64 Hidden Neurons