CAP 6610: Machine Learning – Homework 2

James Diffenderfer

March 15, 2017

Problem 1. Single Node Neural Network.

(a) Implement a Single Node Neural Network using the Perceptron Learning Algorithm we proved in the class. The activation function is the sign function. Use the Bank-Marketing dataset for this part and choose a 50-50 training-testing split.

(b) Use Gradient Descent on a Single Sigmoid Neuron (e.g. $y = \sigma\left(\sum w_i x_i\right)$) instead of the sign function to learn the discriminant. Use the same dataset as in part (a).

Data preprocessing. Before implementing the perceptron learning algorithm or a single sigmoid neuron, I had to edit the data set to convert all of the nominal values to values that could be used as inputs to the learning algorithms for training and testing. For each column in the data set correspnding to a nominal value, I determined the number of possible choices for that column. Then I created a new column for each one of these choices and assigned a 1 if the original data point was classified as this nominal value and a 0 otherwise. For example, the second column of `bank.csv` had the header "job". All of the possible options for this column were: 'admin.', 'blue-collar', 'entrepreneur', 'housemaid', 'management', 'retired', 'self-employed', 'services', 'student', 'technician', 'unemployed', and 'job_unknown'. So I created a new data file with all of these values listed as a header for different columns. If a person had job 'retired' in the original data file `bank.csv`, then in the new data file they had a 1 in the 'retired' column and a 0 in all other columns pertaining to job categories. The routine for performing this operation for the entire data set in `bank.csv` can be found in the file `problem1_data.csv`. This program creates and save the data files `bank_feature.csv` and `bank_target.csv` which contain nominal representations of the feature and target data from `bank.csv`.

Results for (a). For this part I coded the perceptron learning algorithm using python. I initialized each weight and bias term using `numpy.random.uniform(-1, 1)`. For the update of the weights, I used a fixed learning rate of 1. Results from one experiment can be found in Table 1.

Number of Epochs	Misclassified Testing Points (Pre-Training)	Misclassified Testing Points (Post-Training)
100	74.1 %	11.78 %

Table 1: Results from training and testing perceptron learning algorithm over 100 epochs.

Results for (b). For this part of the problem, I coded the single neuron neural network using python. As in part (a), I initialized each weight and bias term using `numpy.random.uniform(-1, 1)`. For the update of the weights, I found that using a full gradient descent took a fair amount of time since the training set had over 2000 rows with 44 columns (after data preprocessing). To decrease the training time I used a batch stochastic gradient descent with batch size 10. Using this method, I noticed an decrease in the amount of time training and the algorithm still achieved good classification results. Results from one experiment can be found in Table 2.

Number of Epochs	Misclassified Testing Points (Pre-Training)	Misclassified Testing Points (Post-Training)
50	80.7 %	15.67 %

Table 2: Results from training and testing single sigmoid neural network over 50 epochs.

Problem 2. Code and test a two layer feed-forward net of sigmoidal nodes with two input units, ten hidden units and one output unit that learns the concept of a circle in 2D space. The concept is:

$$\text{If } (x-a)^2+(y-b)^2 < r^2 \implies \langle x, y\rangle \text{ is labeled } +$$
$$\text{If } (x-a)^2+(y-b)^2 \geq r^2 \implies \langle x, y\rangle \text{ is labeled } -$$

Draw all data from the unit square $[0,1]^2$. Set $a = 0.5$, $b = 0.6$, $r = 0.4$. Generate 100 random samples uniformly distributed on $[0,1]^2$ to train the network using error back-propagation and 100 random samples to test it. Repeat the procedure multiple epochs and with multiple initial weights. Report the changing accuracy and the hyperplanes corresponding to the hidden nodes (when the sigmoid is turned into a step function).

Results. For this problem, I used `numpy.random.uniform(-1, 1)` to generate each term for the initial weights and biases. For the backpropagation phase, I used a stochastic gradient descent to update the weights and I did not experience any trouble converging to local minimum that did a good job classifying the data points. To understand how the number of epochs affected how many points were appropriately classified, I experimented by starting with the same initial weights and training them over various numbers of epochs. Results of one of these experiments can be found in Figure 1, Figure 2, and Table 3.

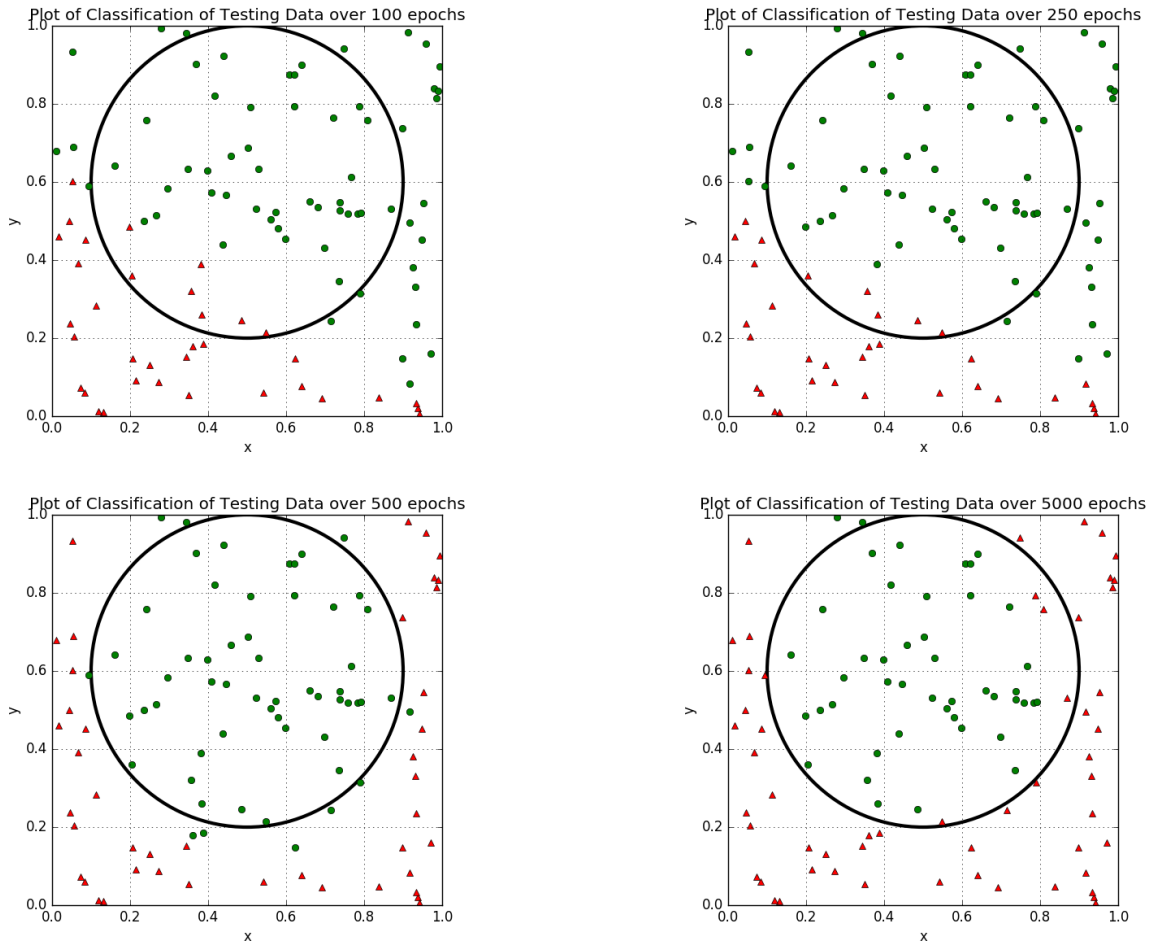

Figure 1: Plots of classification of testing data over 100, 250, 500, and 5000 epochs. In these figures, dots represent the points that were classified as inside the circle and triangles represent the points that were classified as outside of the circle.

As observed in Figure 1, the model begins to classify the points correctly after 500 epochs. The classification errors observed inside of the circle when training over 5000 epochs are due to overfitting of the training data. Error

is also introduced into the model by our choice of data points. Since they are randomly distributed in $[0,1]^2$, there are often regions of the training data near the boundary of the circle that do not contain any data points. This seemed to result in misclassification of data points in the testing set that fall within these areas. After making this observation, I experimented with training and testing on larger data sets taken from $[0,1]^2$ to satisfy my own curiousity and see what kind of effect it would have on the classification.

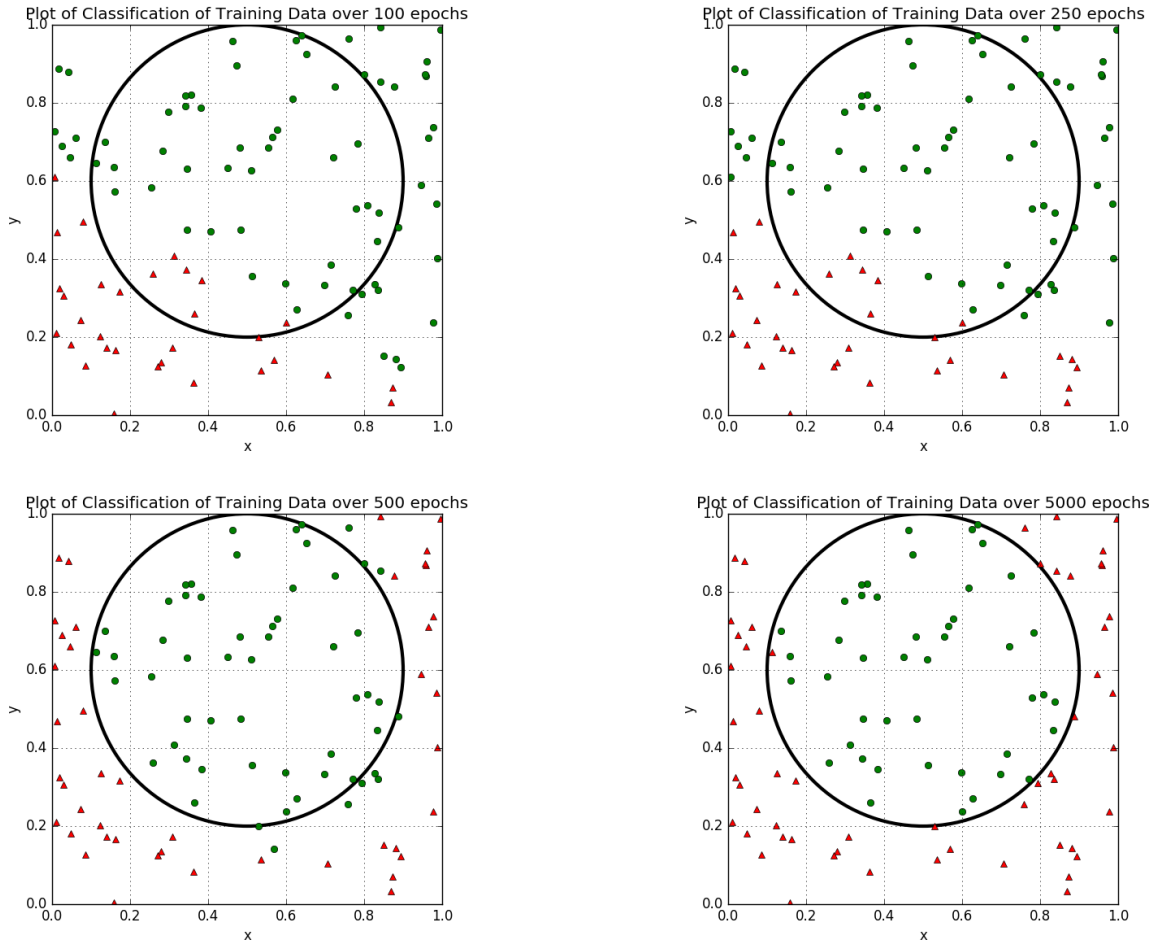

Figure 2: Plots of classification of training data over 100, 250, 500, and 5000 epochs. In these figures, dots represent the points that were classified as inside the circle and triangles represent the points that were classified as outside of the circle.

Number of Epochs	Training Error	Testing Error	Percent of Misclassified Points from Testing Set
0	13.3475946994	13.2513353381	53
100	12.7437237191	11.6489353614	32
250	12.7826655643	11.5108626528	30
500	3.93259352257	3.98997848821	10
2500	1.38300568026	3.32401140422	7
5000	0.42780785905	2.61397288847	6

Table 3: Summary of results from training and testing nerual network over 0, 100, 250, 500, 2500, and 5000 epochs.

Problem 3. Consider the following 2 node Recurrent Neural Network (RNN).Each node receives input from that time step, and the output/state of the other node form the previous time step. Each node has a bias. Your goal is to learn $w_1, \hat{w}_1, b_1, w_2, \hat{w}_2, b_2$ from a given dataset using error back propagation through time. The dataset format is $[x_1, x_2, y_1, y_2]$.

Results. For this problem I coded the recurrent neural network using python. As in previous exercises, I initialized each weight and bias term using `numpy.random.uniform(-1, 1)`. For the backpropagation phase of the algorithm, I first computed the partial derivatives of the total error with respect to $w_1, \hat{w}_1, b_1, w_2, \hat{w}_2, b_2$ in terms of the outputs from each node in the network. Afterwards, I coded routines for computing these given a data point from the feature vector. To perform the updates for the weight and bias terms, I experimented with full gradient descent, batch stochastic gradient descent, and stochastic gradient descent. When using full gradient descent, my algorithm quickly converged to a local minimum of the error function which (very often) resulted in the erro function having a value greater than 23. When using batch stochastic gradient descent, I found that using a smaller batch size resulted in more consistent convergence to a local minimum yielding the value of approximately 0.16 for the error function. Below are the outputs from one experiment of this recurrent neural network trained over 100 epochs:

```
------------------------- Final Weights -------------------------
```

$$\hat{w} = [-0.712475, 0.313653]$$
$$b = [0.200493, -0.913314]$$
$$w = [1.5363, 1.46793]$$

```
-------------------------- Total Error --------------------------
```

$$\frac{1}{2}||t - o||^2 = 0.16026$$

At the time of submission, the best result I can get for my algorithm is a minimum total error of approximately 0.16. I suspect that there is a small typo somewhere in my code but at the time of writing this report I was unable to locate and fix the typo.