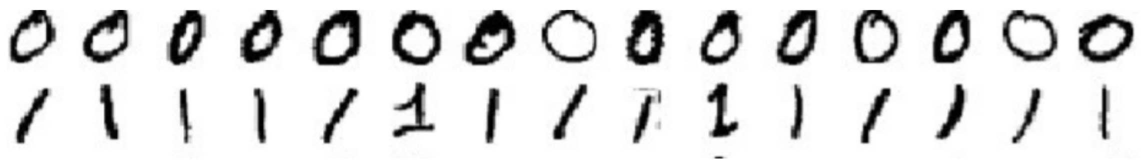
CS 460G Homework #4
Multi-Layer Perceptrons

Steven Penava
03/21/2018

Overview and Instructions

The purpose of this program is to use the MNIST data set to create a multi-layer perceptron to classify images of handwritten 1's or 0's, based on their pixel values, as either a 1 or a 0. The images are presented as such:



Each image of a digit is represented as a 28x28 matrix in the input CSV sets, representing pixel values for each pixel in the matrix on a scale from 0 to 255. In our case, 0 is taken to be blank space or white and 255 is taken to be complete darkness. The program aims to train on the training set to learn what constitutes a 1 and what constitutes a 0, with surprisingly accurate results.

There are three important files in this program:

```
perceptron.py
data/mnist_test_0_1.csv
data/mnist_train_0_1.csv
```

Run `perceptron.py` with the IDE of your choice, using Python 3. Ensure that the `data` folder is in the same directory as the Python script.

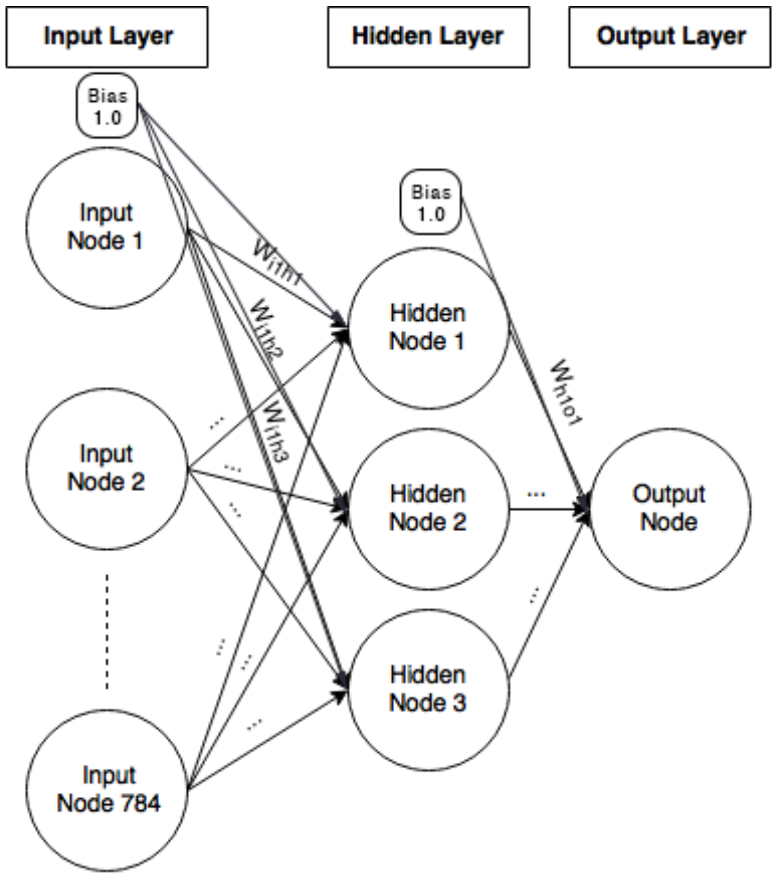
NOTE: when you run the program you may encounter an overflow warning. Ignore this, as it has no effect on the performance.

Implementation and Decisions

When running the Python script, the first thing that will occur is the loading of the CSV files. After this completes, the sequence of epochs begin to run. Each epoch runs the training set (~12,000 iterations) and an error check using the test set (~2,000 iterations). The epochs continue to run until an accuracy rating of at least 99% is achieved.

Depending on the initial weights, this could take a while. I have completed several runs and the accuracy rating starts around 90% for the first epoch, so expect 4-10 runs on average to reach sufficient accuracy.

My program uses one hidden layer with three nodes and one output layer with one node. **I have a bias of 1.0 added to the input and hidden layers.** I did this by inserting 1.0 at the beginning of the input data rows and increasing the size of the weight matrices to compensate. The Sigmoid activation function is still calculated but is overridden to result in 1.0 for the bias nodes. Below is a visualization of what my neural network looks like:



For every epoch, the input matrix (size 785 because of extra bias node) is looped through to update weights with backpropagation according to necessity. My weight matrices are called `weightsIH` and `weightsHO`, meaning “input to hidden” and “hidden to output”, respectively. `weightsIH` is of the form (785, 4), meaning 785 rows by 4 columns (extra row and column due to bias – Sigmoid calculation is overridden for the extra node to be 1). `weightsHO` is of the form (4, 1). Each matrix contains random weights between -1 and 1. These are formed in this way to ensure correct matrix operations are carried out when calculating the dot product. The weights are completely random, so some instances of these weights produce better results more quickly than others (it depends on the initial run of the program). Numpy is used for matrix algebra and storage.

My activation function used is known as the Sigmoid Function. Here is the formula:

$$S(x) = \frac{1}{1 + e^{-x}}$$

In our case, the “x” is a set of inactivated nodes in a specific layer. This is applied to every node in the hidden layer and output layer after a matrix operation is performed. To check for accuracy, I simply kept count of correct guesses with the test set and divided that count by the number of items in the entire set.

Results

Here are the results of two runs:

Run #1 (Accuracy achieved in 4 epochs):

Epoch Iteration #	**Accuracy**
1	94.99%
2	96.50%
3	98.82%
4	99.20%

Run #2 (Accuracy achieved in 6 epochs):

Epoch Iteration #	**Accuracy**
1	86.81%
2	94.04%
3	97.12%
4	98.58%
5	98.87%
6	99.01%