IMPLEMENTATION

In this section I will discuss the implementation of the processes involved in building and testing a neural net (NN) for the given data.

Files

main.py: Contains only main, the driver function for the assignment code. It first requests *trials, hidden nodes, epochs,* and *alpha* as inputs from the user. It runs *trials* iterations of building and testing the NN, then, after these iterations, prints a summary of the average performance of the NN. The code as a whole can be run with the command "*python3 main.py"*.

nnmath.py: Contains the functions used to build and test the NN. The most important among these include rand_weights, forward_pass, and backprop.

csvread.py: Contains the functions used to read in the data from the .csv files given with the assignment. The only function among these used by main is read_norm_mnist. These files must be added by the user, as they were to large to submit.

visualize.py: Contains functions which print the MNIST images read in from the .csv files. These functions are used to print images which were misclassified. The function show_mnist_multiple is run at the end of main should the user ask for this.

**.csv:* Data files given with the assignment. These files are expected to be formatted with the first value of each row being that row's class label, and the 784 following values to be the grayscale pixel values of an MNIST image of a 0 or a 1. As I chose not to implement the code for the bonus portion of the assignment, the only files needed of the four provided are *mnist_train_0_1.csv* and *mnist_test_0_1*.csv.

Miscellaneous Implementation Notes

• Bias is added the first node of the *x* layer which, initially, has a value of 1 and a weight of 1.

• This NN is implemented with one hidden layer *h*, with a user-input number of nodes, and a single output node *o*. The activation (a_o) of the output node determines the output of the network and is rounded: output is thus 1 when $a_o >= 0.5$, and 0 otherwise. In practice, this division at 0.5 is hardly necessary, as outputs are always either extremely close to 1 or 0. The division at 0.5 merely serves as a true catch-all for dividing the outputs.

• The initial weights of the NN are randomly chosen from the continuous interval [-1, 1].

• Each row of read-in data is normalized using the function normalize, which normalizes each row according to the maximum value in that row. That is to say, each value in the row is changed to its proportion of the largest value in the row.

• While the number of nodes, the number of epochs, and the alpha value may all be entered by the user, I have discussed my preferred values for these in the results section at the end of this report.

• As this code relies on the python numpy library, the user must have it installed in order to run *main.py*. Should it not be installed, it can be installed with the command *"pip install numpy"*.

Matrices

The vast majority of mathematical operations performed within the code is performed using numpy library matrices. This is contrast to the creation and use of a node class, as matrix-based operations are faster and simpler. For this report, consider the following variables to represent the following:

• x represents the normalized grayscale pixel values of one MNIST image. That is to say, the values of the nodes at the input layer. In the code, it is represented as a column vector with a height equal to the number of pixel values plus one bias term.

• w_1 represents the weights from the input layer to the hidden layer h. In the code, it is stored as a matrix with as many rows as there are inputs and as many columns as there are nodes in h. This is so that w_1 may be multiplied with x.

• w_2 represents the weights from h to the output layer o. In the code, it is stored as a column vector with a height equal to the number of hidden nodes. This is so that it may be multiplied with a row vector representing the activations of the nodes in h.

• a_h represents the activations of nodes in h. In the code, it is stored as a row vector with length equal to the number of nodes in h.

• a_o represents the activation of nodes in the o layer. As this layer is only ever 1 node, this is stored as a 1x1 matrix.

Activation

The activation of a node is calculated using the sigmoid function, the value of which is returned by sig. This value is:

$$\sigma(n) = \frac{1}{1 + e^n}$$

Forward Pass

The function forward_pass calculates and returns a_h, a_o using the activations of the input layer and w_1, w_2. This amounts to an application of the sigmoid function on each value in the matrices yielded by w_1x and w_2a_h to calculate a_h and a_o, respectively.

Backpropogation

The function backprop serves to implement the backpropogation algorithm, which updates w_1 and w_2. This is performed according to the following equations:

$$\sigma'(a_o) = \sigma(a_o)\big(1 - \sigma(a_o)\big)$$

$$\Delta_o = Error_{output} \times \sigma'(a_o), \qquad w_{h_i,o} = w_{h_i,o} + \alpha\Delta_o a_{h_i}$$

$$\Delta_h = \sigma'(a_h) \times w_2\Delta_o, \qquad w_{x_j,h_i} = w_{x_j,h_i} + \alpha\Delta_{h_i} a_{x_j}$$

Testing Correctness

Testing the accuracy of the NN's final weights, decided after *epochs* epochs have been completed, is done by comparing the activation of *o,* produced after a forward pass using each image's data and the final weights, to the label of that image. This testing is performed within main after the NN is built.

The results of this testing with varied inputs are documented at the end of this report.

Visualization

Though it is not required, the code also contains functions for visualization. The function show_mnist serves to display a single MNIST image from its data row, and the function show_mnist_multiple serves to do this for multiple MNIST images, displaying them in a grid. This code is included for viewing images which were misclassified by the NN, largely for curiosity purposes but also for determining obvious issues with the NN.

RESULTS

Below is a table of the results of the NN with varied user inputs. All rows are run at 30 trials, as this is typically the rough minimum number said by the central limit theorem and law of large numbers to be the sample size at which the sample mean approaches the true mean. It should also be noted that the runtime of one single trial is higher than the average runtime of multiple trials (as output by main's summary) with the same settings due to python's caching. Note that all of the values presented here are rounded to four decimal places. While the code does use rounded values for calculations, I have rounded them here and in the summaries of main for readability.

From my experimentation with this implementation, it appears, as expected, that increasing *hnodes* or *epochs* has a negative effect on the runtime—*hnodes* more so—and increasing *alpha* has little effect. Increases to *hnodes* also appears to have the most meaningful affect on the accuracy of the NN.

Additionally, it appears that 5 hidden nodes is the minimum number of nodes required to consistently reach the benchmark 80% accuracy. However, 80% is rather low for real application. Instead, I find the highlighted row to be a good balance of speed, accuracy, and consistency. Were I to remove the user-input fields of main, instead hard-coding values of *trials, hnodes, epochs,* and *alpha*, the values within the highlighted row are the values I would use.

Using these values, building the NN once typically takes about 2 minutes on my hardware, and typically yields an accuracy just above 99%. This translates to approximately 20 or less mislabeled images. On the next page is an example of the images misclassified by a NN built with these settings, as output by show_mnist_multiple.

TRIALS	H NODES	EPOCHS	ALPHA	MEAN AVG ACCURACY (%)
30	4	10	1	77.1757
30	4	10	0.1	77.9921
30	4	10	0.01	79.8006
30	4	100	1	81.0197
30	4	100	0.1	84.0835
30	4	100	0.01	80.818
30	5	10	1	84.7675
30	5	10	0.1	81.9795
30	5	10	0.01	83.7967
30	5	100	1	85.1868
30	5	100	0.1	86.0441
30	5	100	0.01	84.2301
30	16	10	1	97.4043
30	16	10	0.1	97.4806
30	16	10	0.01	96.1718
30	16	100	1	97.7715
30	16	100	0.1	97.0134
30	16	100	0.01	96.0227
30	32	10	1	99.3012
30	32	10	0.1	99.3885
30	32	10	0.01	99.2993