2802ICT Intelligent Systems, Assignment #2, Trimester 1, 2017

Python Neural Network

Written by Jack Kearsley (s5007230)
jack.kearsley@griffithuni.edu.au

Table of Contents

Introduction

The Artificial Neural Network is a machine learning model that is able to predict and classify input data, through the utilisation of sets of ‘neurons’ connected by weighted synapses. The model is loosely based off the neurons and axons found in biological brains, and has become popular in recent years due to an increase in computing power and an abundance of data.

For this assignment, the task is to implement a neural network in Python that is capable of classifying handwritten digits sourced from the MNIST database. The digits are stored as small gray scale images with dimensions of 28 by 28 pixels. The network must be trained and tested with the provided data sets (TrainDigitX.csv.gz, TrainDigitY.csv.gz, TestDigitX.csv.gz, TestDigitX2.csv.gz, TestDigitY.csv.gz).

Part 1: Manual Calculation for a Small Neural Network

Below are the manual calculations for various values of a small neural network, consisting of 2 input neurons, 2 hidden neurons, and 2 output neurons. There are two samples, X_1 = (0.1, 0.1) and X_2 = (0.1, 0.2), with corresponding Y values of Y_1 = (1, 0) and Y_2 = (0, 1), respectively. The weights are updated once using stochastic gradient descent with backpropagation. The batch-size is 2, the number of epochs is 1, and the learning rate (η) is 0.1.

Manual Calculations

$$z^2 = X^T W^1$$

$$\begin{bmatrix} x_1^a & x_1^b & b_1 \\ x_2^a & x_2^b & b_2 \end{bmatrix} \begin{bmatrix} w_1^1 & w_2^1 \\ w_3^1 & w_4^1 \\ w_9^1 & w_{10}^1 \end{bmatrix} = \begin{bmatrix} z_{1a}^2 & z_{1b}^2 \\ z_{2a}^2 & z_{2b}^2 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.1 & 1 \\ 0.1 & 0.2 & 1 \end{bmatrix} \begin{bmatrix} 0.1 & 0.2 \\ 0.1 & 0.1 \\ 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.12 & 0.13 \\ 0.13 & 0.14 \end{bmatrix}$$

$$a^2 = f(z^2)$$

$$a^2 = \begin{bmatrix} f(z_{1a}^2) & f(z_{1b}^2) \\ f(z_{2a}^2) & f(z_{2b}^2) \end{bmatrix} \quad f(x) = \frac{1}{(1 + e^{-x})}$$

$$a^2 = \begin{bmatrix} 0.5300 & 0.5325 \\ 0.5325 & 0.5349 \end{bmatrix}$$

$$z^3 = (a^2)^T W^2$$

$$\begin{bmatrix} a_{1a}^2 & a_{1b}^2 & b_1 \\ a_{2a}^2 & a_{2b}^2 & b_2 \end{bmatrix} \begin{bmatrix} w_5^2 & w_6^2 \\ w_7^2 & w_8^2 \\ w_{11}^2 & w_{12}^2 \end{bmatrix} = \begin{bmatrix} z_{1a}^3 & z_{1b}^3 \\ z_{2a}^3 & z_{2b}^3 \end{bmatrix}$$

$$\begin{bmatrix} 0.5300 & 0.5325 & 1 \\ 0.5325 & 0.5349 & 1 \end{bmatrix} \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.2063 & 0.2595 \\ 0.2067 & 0.2602 \end{bmatrix}$$

$$y^{hat} = \begin{bmatrix} f(z_{1a}^3) & f(z_{1b}^3) \\ f(z_{2a}^3) & f(z_{2b}^3) \end{bmatrix}$$

$$y^{hat} = \begin{bmatrix} 0.5514 & 0.5645 \\ 0.5515 & 0.5647 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\delta^3 = -(y - y^{hat}) * f'(z^3) \qquad f'(x) = x(1-x)$$

$$\delta^3 = -(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0.5514 & 0.5645 \\ 0.5515 & 0.5647 \end{bmatrix}) * \begin{bmatrix} 0.2474 & 0.2458 \\ 0.2473 & 0.2458 \end{bmatrix} = \begin{bmatrix} -0.1110 & 0.1388 \\ 0.1364 & -0.1070 \end{bmatrix}$$

$$\frac{dE}{dW^2} = (a^2)^T \delta^3$$

$$\frac{dE}{dW^2} = \begin{bmatrix} 0.5300 & 0.5325 \\ 0.5325 & 0.5349 \end{bmatrix} \begin{bmatrix} -0.1110 & 0.1388 \\ 0.1364 & -0.1070 \end{bmatrix} = \begin{bmatrix} 0.01380 & 0.01658 \\ 0.01385 & 0.01667 \end{bmatrix}$$

$$\frac{dE}{db^2} = (b^2)^T \delta^3$$

$$\frac{dE}{db^2} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} -0.1110 & 0.1388 \\ 0.1364 & -0.1070 \end{bmatrix} = \begin{bmatrix} 0.0254 & 0.0318 \end{bmatrix}$$

$$\delta^2 = \delta^3 (W^2)^T * f'(z^2)$$

$$\delta^2 = \begin{bmatrix} -0.1110 & 0.1388 \\ 0.1364 & -0.1070 \end{bmatrix} \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \end{bmatrix} * \begin{bmatrix} 0.2491 & 0.2489 \\ 0.2489 & 0.2488 \end{bmatrix} = \begin{bmatrix} 0.00069 & 0.00416 \\ 0.00072 & -0.00194 \end{bmatrix}$$

$$\frac{dE}{dW^1} = X^T \delta^2$$

$$\frac{dE}{dW^1} = \begin{bmatrix} 0.000141 & 0.000222 \\ 0.000213 & 0.000028 \end{bmatrix}$$

$$\frac{dE}{db_1} = (b^1)^T \delta^2$$

$$\frac{dE}{db_1} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 0.00069 & 0.00416 \\ 0.00072 & -0.00194 \end{bmatrix} = \begin{bmatrix} 0.00141 & 0.00222 \end{bmatrix}$$

$$\eta = 0.1$$

$$W^1_{new} = W^1 - \eta \frac{dE}{dW^1} \qquad W^1_{new} = \begin{bmatrix} 0.09998 & 0.19997 \\ 0.09997 & 0.099997 \end{bmatrix}$$

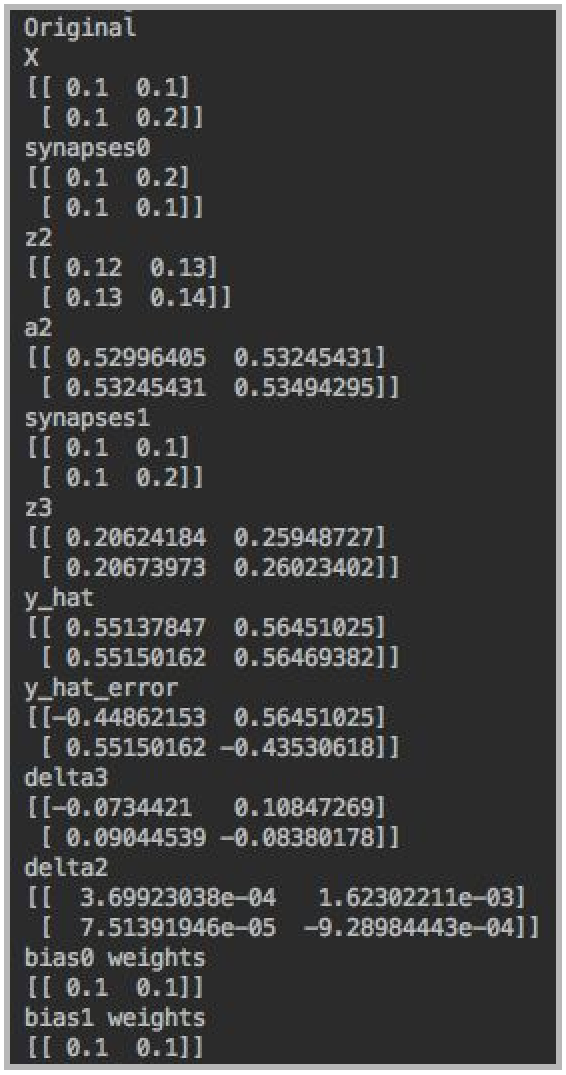
$$W^2_{new} = W^2 - \eta \frac{dE}{dW^2} \qquad W^2_{new} = \begin{bmatrix} 0.09862 & 0.0983 \\ 0.098615 & 0.1983 \end{bmatrix}$$

$$b^1_{new} = b_1 - \eta \frac{dE}{db_1} \qquad b^1_{new} = \begin{bmatrix} 0.099859 & 0.099778 \end{bmatrix}$$

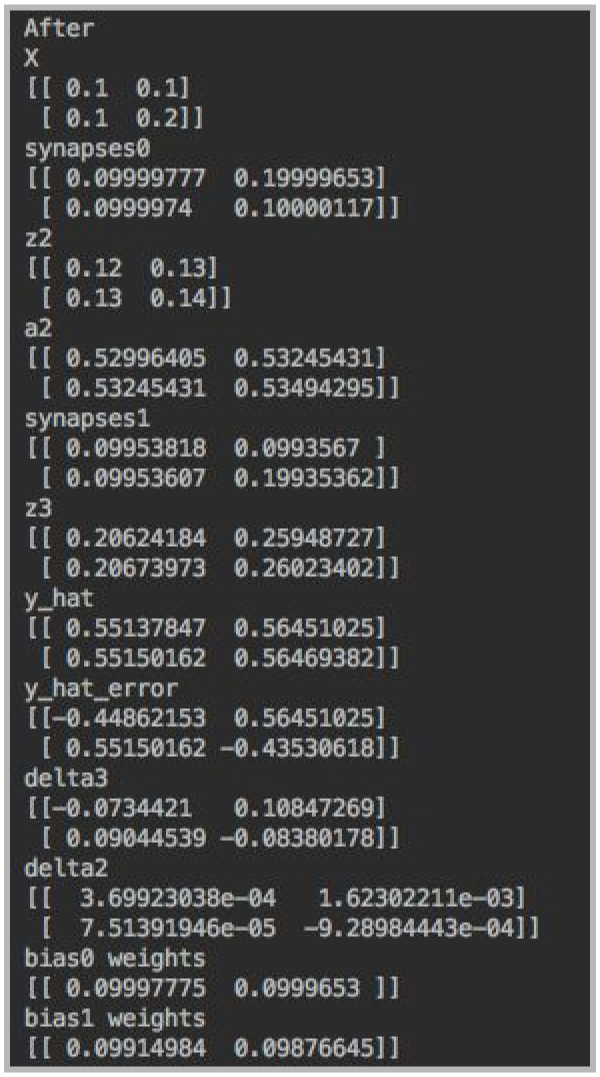
$$b^2_{new} = b_2 - \eta \frac{dE}{db_2} \qquad b^2_{new} = \begin{bmatrix} 0.09746 & 0.09682 \end{bmatrix}$$

Validation Using Implemented Neural Network

To verify that the above calculations are correct, this small neural network was tested with the neural network program implemented in Python. The X, Y, mini-batch-size, learning rate, and weight values were coded into the program manually using numpy arrays. Before any backpropagation (Original), and after the same number of epochs (1) used in the hand-calculations, the values for the relevant matrices are displayed below.



```
Original
X
[[ 0.1  0.1]
 [ 0.1  0.2]]
synapses0
[[ 0.1  0.2]
 [ 0.1  0.1]]
z2
[[ 0.12  0.13]
 [ 0.13  0.14]]
a2
[[ 0.52996405  0.53245431]
 [ 0.53245431  0.53494295]]
synapses1
[[ 0.1  0.1]
 [ 0.1  0.2]]
z3
[[ 0.20624184  0.25948727]
 [ 0.20673973  0.26023402]]
y_hat
[[ 0.55137847  0.56451025]
 [ 0.55150162  0.56469382]]
y_hat_error
[[-0.44862153  0.56451025]
 [ 0.55150162 -0.43530618]]
delta3
[[-0.0734421   0.10847269]
 [ 0.09044539 -0.08380178]]
delta2
[[  3.69923038e-04   1.62302211e-03]
 [  7.51391946e-05  -9.28984443e-04]]
bias0 weights
[[ 0.1  0.1]]
bias1 weights
[[ 0.1  0.1]]
```



```
After
X
[[ 0.1  0.1]
 [ 0.1  0.2]]
synapses0
[[ 0.09999777  0.19999653]
 [ 0.0999974   0.10000117]]
z2
[[ 0.12  0.13]
 [ 0.13  0.14]]
a2
[[ 0.52996405  0.53245431]
 [ 0.53245431  0.53494295]]
synapses1
[[ 0.09953818  0.0993567 ]
 [ 0.09953607  0.19935362]]
z3
[[ 0.20624184  0.25948727]
 [ 0.20673973  0.26023402]]
y_hat
[[ 0.55137847  0.56451025]
 [ 0.55150162  0.56469382]]
y_hat_error
[[-0.44862153  0.56451025]
 [ 0.55150162 -0.43530618]]
delta3
[[-0.0734421   0.10847269]
 [ 0.09044539 -0.08380178]]
delta2
[[  3.69923038e-04   1.62302211e-03]
 [  7.51391946e-05  -9.28984443e-04]]
bias0 weights
[[ 0.09997775  0.0999653 ]]
bias1 weights
[[ 0.09914984  0.09876645]]
```

These computed values are almost identical to the hand-calculated values, hence proving that the latter are correct. The slight differences can be attributed to the rounding of the hand-calculated numbers for simplicity.

Part 2: Implementation in Python

User Instructions

- The neural network learning algorithm was implemented using Python (version 3.6), and is capable of classification. The source code for the program is contained in the file 'neural_network.py', and accepts seven arguments (NInput NHidden NOutput TrainDigitX.csv.gz TrainDigitY.csv.gz PredictDigitY.csv.gz). The meaning of these arguments is outlined on the assignment sheet.
- The user must have the 'NumPy' package installed.
- The user must import all of the 'csv.gz' files (except for PredictDigitY.csv.gz) into the 'csv-input' directory, located within the program's root directory.
- The neural network's predictions for a given test set (ie. TestDigitX.csv.gz) will be outputted to a file specified by the 'PredictDigitY.csv.gz' argument, into the 'csv-output' directory.

Results

The implemented neural network achieved an average accuracy rate of **95.44%** when classifying data from the 'TestDigitX.csv.gz' test set. The optimal parameters were concluded to be:

Epochs	**Mini-batch Size**	**Learning Rate**	**Bias Input**
30	20	3.0	0.0

These values were decided upon after the analysis of several tests performed on the neural network. Each test varied a certain parameter in order to gain insights into the parameter's effect on classification accuracy. The tests are displayed below, and grouped by parameter type.

Epochs

The graph in Figure 2.1 displays the accuracy of the classifications for TestDigitX.csv.gz, with varied epoch numbers (from 3 to 90). It is clear that an epoch value below 20 produces sub-optimal results, and this is because the neural network requires several epochs to adjust the synapse weights. However, the increase in accuracy obtained by adding more epochs reaches a plateau after 30 epochs. Thus, the optimal number of epochs was chosen to be 30, as any additional epochs after this point result in no significant accuracy gains and slower time-efficiency (see Figure 2.2).

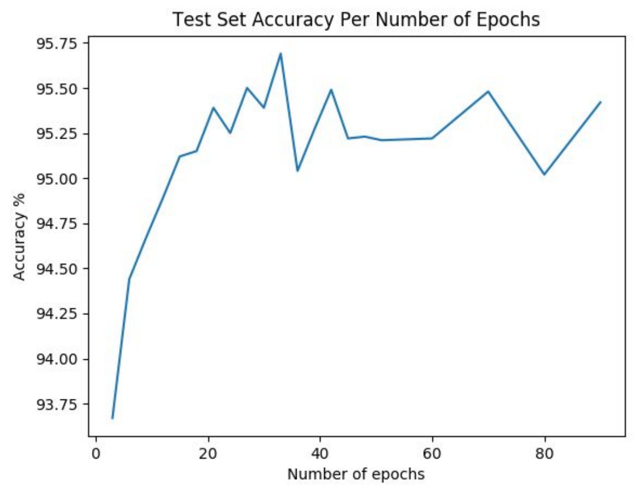


Figure 2.1: Test Set Accuracy Per Number of Epochs.

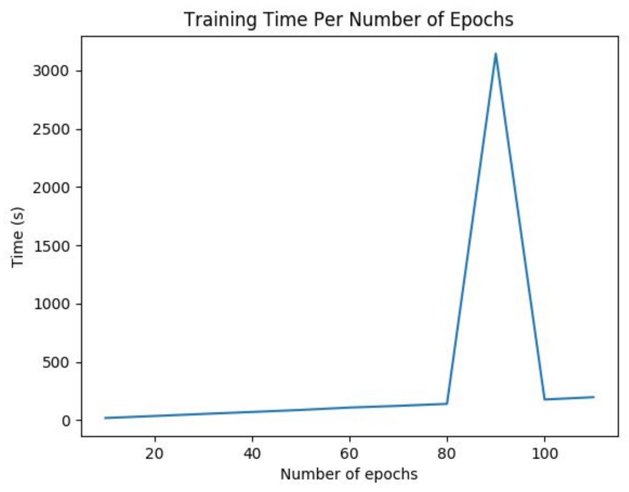


Figure 2.2: Training Time Per Number of Epochs.

Mini-batch Size

The graph in Figure 2.3 displays the accuracy of the classifications for TestDigitX.csv.gz, with varied mini-batch sizes (from 5 to 100). It is clear from Figure 2.3 that mini-batch sizes < 10 and > 35 yield sub-optimal accuracy results. The time taken to train the neural network, for each mini-batch size was also measured, and this data is displayed in the graph from Figure 2.4. The relationship between mini-batch size and training time is linear, and therefore, smaller mini-batch sizes were faster to train on my neural network. In light of this data, the best mini-batch size was chosen to be 20, as this value lies between 10 and 35, and is faster to train than larger values. While this value is optimal for the size of the given data sets, it should be noted that other data sets may require different values for the mini-batch size to produce the highest accuracy possible.

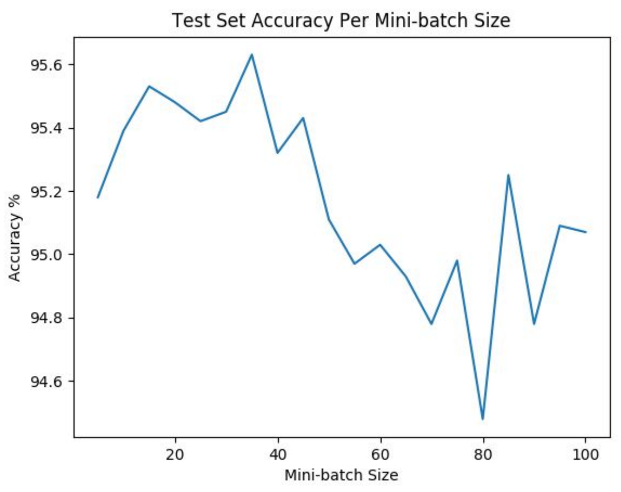


Figure 2.3: Test Set Accuracy Per MIni-Batch Size.

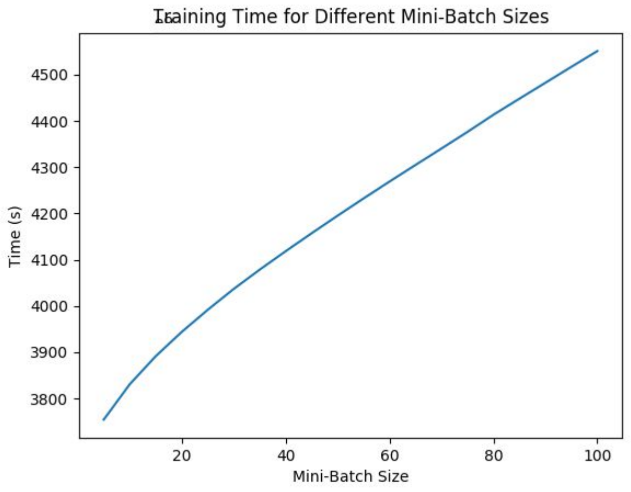


Figure 2.4: Training Time for Different Mini-Batch Sizes.

Learning Rate

Different learning rate values from [10^{-3}..10^{2}] were tested on the neural network, and these values, along with their corresponding accuracy results are displayed on the graph below (Figure 2.5). Learning rate values below 10^{-2}, and above 15 (Figure 2.6) clearly produce sub-optimal results. The learning rate of 3.0 was chosen to be used in the final program in light of these results.

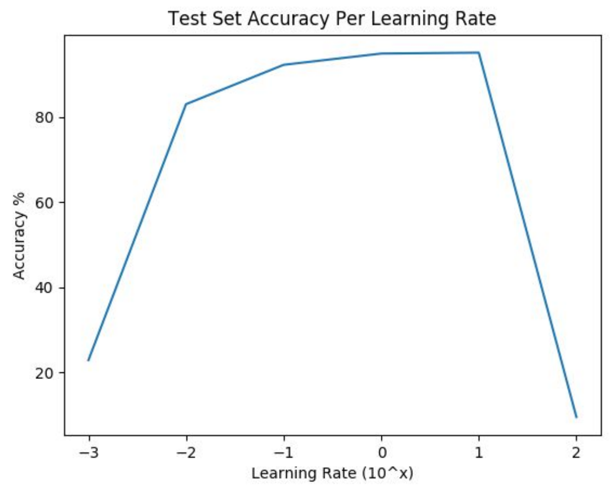


Figure 2.5: Test Set Accuracy Per Learning Rate [10^{-3}..10^{2}]

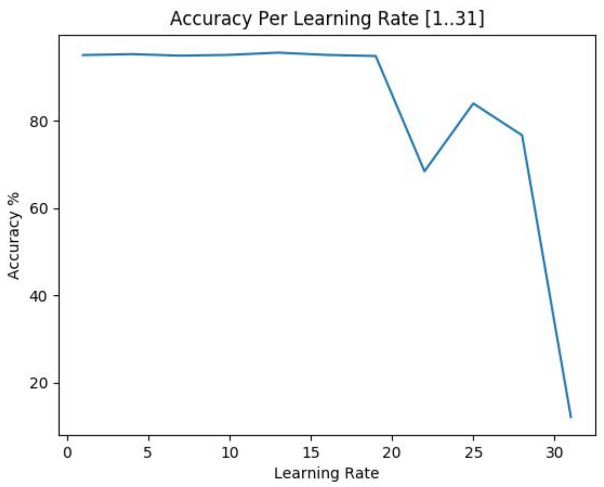


Figure 2.6: Test Set Accuracy Per Learning Rate [1..31]

Bias Input Values

The bias inputs of a neural network are implemented for cases where a neuron's output must be highly activated, even if it's original data inputs are very small or 0. From the graphs below (Figure 2.7 and Figure 2.8), it can be seen that the presence of bias inputs were detrimental to the accuracy of the neural network. Several bias input values from [-1.0…1.0] were experimented with, and ultimately, a bias value of 0 (or essentially no bias inputs at all) yielded the best accuracy. The inputs, synapses, and stochastic gradient descents for bias inputs remain implemented in the program code so the user may experiment with their own values (through the 'bias_num' variable).

Most neural networks require a bias input to produce outputs of 1 when most of the inputs are small or 0. However, the particular data sets used on this neural network (MNIST handwritten digits) are an exception to this rule. Each data sample contains a handwritten digit, represented as grayscale pixel intensity values from 0 (white) to 1 (black). Because each sample is guaranteed to contain a digit (there are no samples that are all 0s and completely white), it can be inferred that some of the input neurons must contain positive activations (ie. > 0). Therefore, it is hypothesised that the bias term becomes somewhat redundant for this data set, and when added into the neural network, causes too much disruption, as it represents another full-black (1) pixel that is not present in the original image.

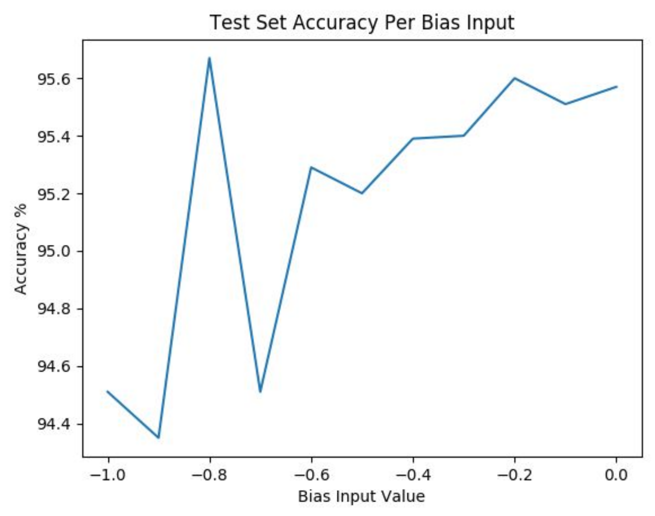
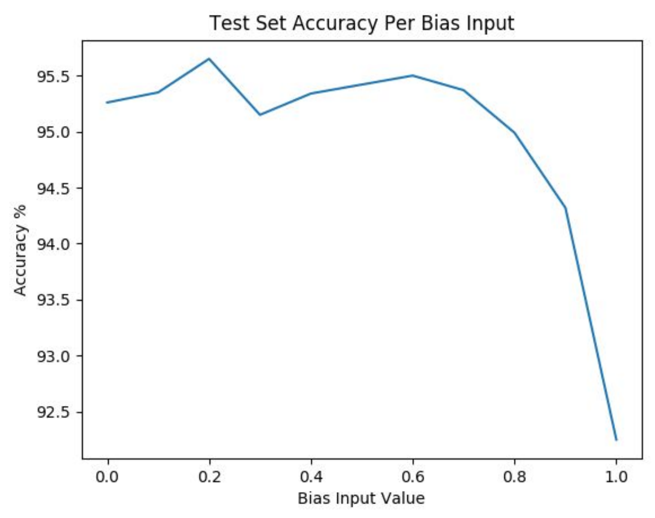
	

Figure 2.7: Test Set Accuracy Per Bias Input [-1..0]	*Figure 2.8: Test Set Accuracy Per Bias Input [0..1]*

Part 3: Alternate Cost Function

For the implementations above, the cost of the neural network's outputs were calculated using the quadratic cost function below:

$$C(w,b) = \frac{1}{2n}\sum_{i=1}^{n} ||f(x_i) - y_i||^2$$

Figure 3.1: Quadratic cost function.

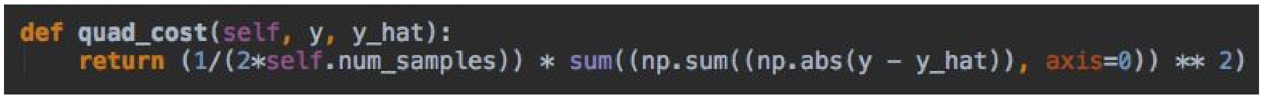
The quadratic cost function was replaced with the cross-entropy cost function, and then the program's classification accuracy on the test data was calculated again, using the same specifications as in Part 2 (epochs = 30, mini-batch size = 20, learning rate = 3.0, bias input = 0.0). The cross-entropy cost function used is displayed below:

$$C(w,b) = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln[f(x_i)] + (1 - y_i)\ln[1 - f(x_i)]$$

Figure 3.2: Cross-entropy cost function.

Implementation

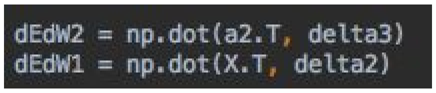
The quadratic cost function from Figure 3.1 was implemented in Python using the function in Figure 3.3.



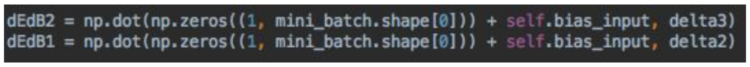
```
def quad_cost(self, y, y_hat):
    return (1/(2*self.num_samples)) * sum((np.sum((np.abs(y - y_hat)), axis=0)) ** 2)
```

Figure 3.3: Quadratic cost function implemented in Python.

Backpropagation using the quadratic cost function (Figure 3.4) was applied using the same formulas found in Part 1.



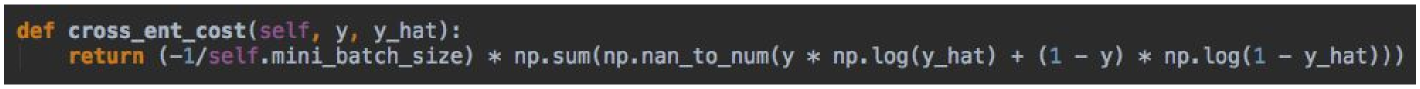
```
dEdW2 = np.dot(a2.T, delta3)
dEdW1 = np.dot(X.T, delta2)
```



```
dEdB2 = np.dot(np.zeros((1, mini_batch.shape[0])) + self.bias_input, delta3)
dEdB1 = np.dot(np.zeros((1, mini_batch.shape[0])) + self.bias_input, delta2)
```

Figure 3.4: Backpropagation using the quadratic cost function.

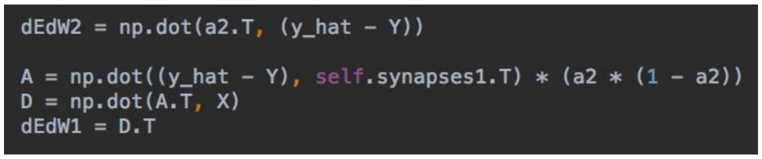
The cross-entropy cost function (Figure 3.2) was implemented in Python using the function in Figure 3.5.



```
def cross_ent_cost(self, y, y_hat):
    return (-1/self.mini_batch_size) * np.sum(np.nan_to_num(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat)))
```

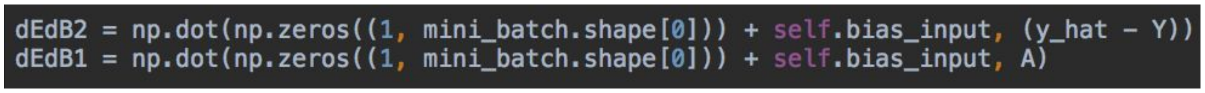
Figure 3.5: Cross-entropy function implemented in Python.

The backpropagation derivatives calculated using the cross-entropy function are different from the quadratic cost backpropagation, so the change in error with respect to W1 and W2 were calculated differently, as seen in Figure 3.6.



```
dEdW2 = np.dot(a2.T, (y_hat - Y))

A = np.dot((y_hat - Y), self.synapses1.T) * (a2 * (1 - a2))
D = np.dot(A.T, X)
dEdW1 = D.T
```



```
dEdB2 = np.dot(np.zeros((1, mini_batch.shape[0])) + self.bias_input, (y_hat - Y))
dEdB1 = np.dot(np.zeros((1, mini_batch.shape[0])) + self.bias_input, A)
```

Figure 3.6: Backpropagation using the cross-entropy cost function.

Results

The use of the cross-entropy cost function resulted in a relatively similar average accuracy of 95.53%.
Figure 3.7 illustrates that the cross-entropy backpropagation reaches a classification accuracy of 95.5% after roughly 20..30 epochs.
The average training time for 20 epochs using the quadratic cost formula was 34.62s, with an average classification accuracy of 95.40%.
The average training time for 20 epochs using the cross-entropy cost formula was 32.91s, with an average classification accuracy of 94.91%.
These results illustrate that the quadratic cost function trains the network with slightly more accuracy (0.49%) than the cross-entropy function, with a trade-off of an additional 1.71s of training time.
The optimal learning rate for the cross-entropy function was found to be 1.0 from the data displayed in Figure 3.8.
In light of this data, the final neural network predictions (for TestDigitX2.csv.gz) were made using the quadratic cost function, as opposed to the cross-entropy function, due to its increased classification accuracy.

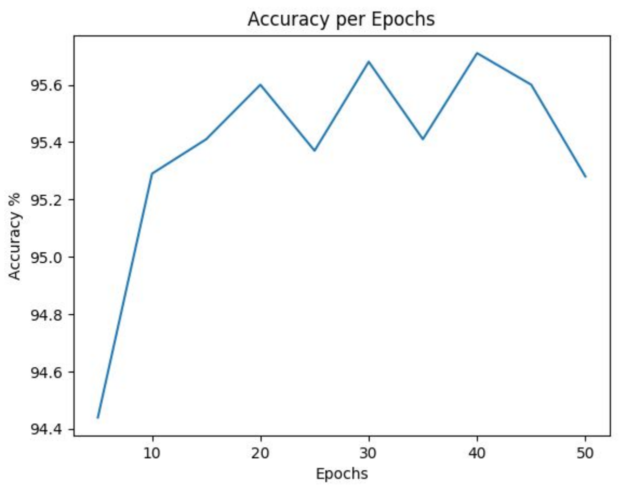


Figure 3.7: Accuracy per Number of Epochs, with learning rate of 1.0 (Cross-entropy).

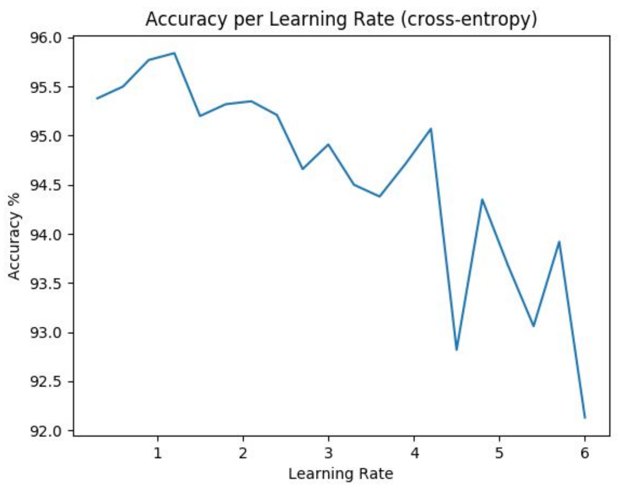


Figure 3.8: Accuracy Per Learning Rate (Cross-entropy).