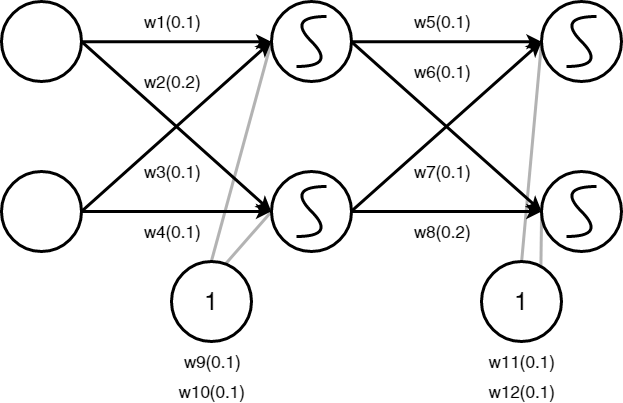
# Simple Neural Network

## Problem Statement

This assignment is based on implementing a machine learning algorithm, a Neural Network using Python. Datasets are provided to train and test the neural network which will be experiemented with different settings of hyper parameters. The datasets are based on the MNIST hand written digits database.

## Part 1 – Manual Calculation for small neural network

Figure 1a – Neural Network



Training samples

Label for

Label for

Firstly, the test to see what the neural network currently predicts. Given the weights and bias above and training inputs this test can be done.

This is done by finding the net input to each hidden layer neuron, squashing the total net input using a logistic function then repeating this for the output later neurons. Using backprobagation each weight can be changed.

To make it more readable, each layer will be named like so:

Finding net input for hidden layer 1 or :

Logistic function to get output:

Finding net input for hidden layer 2 or

Logistic function to get output:

Using the output from the hidden layer neurons, we can use them as inputs for output layer neurons.

Using the first output neuron

Repeated for second output

Using the sqaured error function the error can be calculated, then by summing them the total error can be found.

The target output for is 1, but the neural network output is

The target output for is 0, but the neural network output is

Total error

The weights in the network need to be updated so that they cause the actual output to be closer to the target output, therefore minimising error.

For example, using the amount of change in the weight can affect the total error (the partial derivative of with respect to ):

Using the chain rule formula this can be done:

Finding the first equation, quadratic cost

The next part is finding how much does output with respect to its total net

The final part of the equation, finding how much does the total net input of change with respect to

Now that each piece has been found adding the formula together:

Now to decrease the error, this value is subtracted from the current weight and multiplied by the learning rate (which is 0.1).

New weights for output layer:

For updating the weights on the hidden layer, the same process is applied but the overall formula is slightly altered:

First part of the equation:

The same process is applied but instead it’s used for the second output, because these are redundant caluclations, they were left out. The final calculation:

Therefore, to finish this first equation:

The second part of the equation now needs to be found:

The same process is applied, calculate the partial derivative of the total net input, but instead use with respect to :

Now all the parts have been calculated and the forumla can be put together:

New weights for hidden layer:

Now all the weights are updated.

The rule for bias weights is very similar, expect that there is no input from a previous layer. Instead bias is caused by input from a neuron with a fixed activiation of 1.

## Part 1 – Python Implementation

To start off, the data needs to be split so that there are training, validation and testing sets. By restructuring the way the data sets are stored to be compatible with the interface for the neural network allows better implementation methods. The input will be tuples which are x,y where x is the input and y is the correct output. The output will be encoded (a number) between 0 and 9 by using 10 output neurons. The neuron with the highest activation will be used as the prediction (p) of the network. The output number y is represented as a list of 10 numbers, all of them are 0 except for the correct digit.

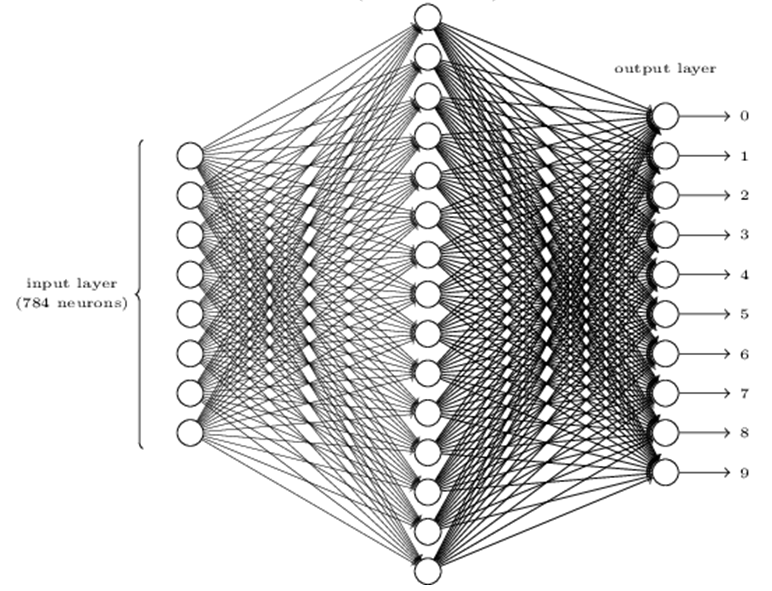


Figure 1b – Neural Network

Figure 1b shows what the neural network may look like in this example (obviously the 784 input neurons have been omitted). 784 input neurons are needed because the handwritten digits are contained in a 28x28=784 pixel image. The output layer contains 10 output neurons corresponding to a digit (number predicition). If the first neuron fires then that will show that the neural network thinks the digit is a 0.

The next part of code involves the sigmoid function and its derivative. The way the derivative was defined was *s(input)\*(1-s(input))* another way of defining that could be computationally superior would be as *input\*(1-input)* then by calling this only with the activation of neurons (which saves on computation of s(input)).

The two cost methods are then defined after the sigmoid function, which are the quadratic and cross entropy cost functions.

The cost function takes the output of the network and compares it to the targets for the equivalent inputs. If the projected outputs differ from the targets, the cost is to be large. The backpropagation algorithm used to train the network is based on partial derivatives of the cost function with respect to the biases and weights of the network. The classes have a delta method which is combining the first terms in the chain rule form for the backpropagation algorithm. The forwardinput function simply takes an input and returns the corresponding output.

The centerpiece is the neuralnetwork class which is representing the entire neural network. It contains a list called ‘body’, which is the number of neurons in the respective layer. Therefore when creating a network the following can be done:  
n = neuralnetwork([784, 30, 10])   
The biases and weights are all initialised randomly using the numpy.random.normal function which generates Gaussian distributions with a mean 0 and standard deviation of 1. This gives the SGD (stochastic gradient descent) algorithm a starting place. The biases and weights are stored as lists of np matrices.  
  
The SGD function takes training data (dt) which is a list of tuples representing the training inputs and preferred outputs. The two variables epochs and batchSize (mini-batch size) are defining the number of epochs to train for and the size of the mini-batches to use. An additional option that was added was the test\_set parameter. Currently it is set to test\_set=none, however if supplied it will evaluate the network of each epoch of training and print out the progress. Which can become useful when tracking progress.  
How this function works is that it starts by randomly shuffling the training set and partitions it into mini-batches, which is a simply way of sampling from the training set. For each mini-batch a single step of gradient descent is applied. This in term updates the network weights and biases according to the single iteration of gradient descent, using just the training data.   
The backprobagate function returns a tuple representing the gradient for the cost function and *dcdw* and *dcdb* are layer-by-layer lists of numpy arrays.

It computes the gradient of the cost function for a training example. By combining backpropagation and stochastic gradient descent, the gradient for many training examples can be done. The general process is as follows:  
1. Input training examples  
2. Set corresponding input activation to perform (for each training example):

1. Feedforward
2. Output error
3. Backpropagate error

3. Gradient descent

A potential modified approach to the backpropagation algorithm would be to compute the gradients for the training examples simultaneously in a mini-batch. Instead of beginning with a single input vector, a matrix can be used instead where the columns are the vectors within the mini-batch. Forward-propagate could be done by multiplying by the matrices weight and applying the sigmoid function.

## Changing Learning Rate

Using a neural network size of [784, 30, 10], tests were undertaken where the learning rate was changed. The parameters for each test were epoch = 30 and minibatch-size = 20 whilst the learning rates were changed to = 0.001, 0.1, 1.0, 10, 100 for each test.

Below is a graph showing the test accuracy based on overall percentage versus Epoch test at different learning rates

**Figure 2**:

An interesting part about this is that the learning rate of 100 stayed consistent throughout the entire test. The accuracy stayed at 10.09% the entire time, whilst the learning rate of 0.001 had the most dramatic change in accuracy from start to finish, going from 8.69% to 55.23% within the final iteration.   
Below is a table showing the maximum accuracy achieved within each learning rate group:

|  |  |
| --- | --- |
| **Overall Maximum Accuracy** | **Learning rate** |
| 55.23% | 0.001 |
| 94.55% | 0.1 |
| 95.83% | 10 |
| 96.45% | 1.0 |
| 10.09% | 100 |

The learning rate of 1.0 yeilds the highest accuracy obtained at 96.45% which shows great results.

Figure 3 shows a single test case. The learning rate was set to 3.  
**Figure 3:**

This was plotted by itself to gain a grasp on the change over time more visually.

## Changing mini-batch size

Using a neural network size of [784, 30, 10], tests were undertaken where the mini-batch size was changed. The parameters for each test were epoch = 30 and = 3.0 whilst the minibatch-sizes were 1, 5, 10, 20, 100 for each test.

Figure 4 shows the accuracy achieved for each mini-batch size of 1, 5, 10, 20 and 100. It can be seen that the size of 1 has interesting behaviour whilst the others seem to be within the same group and range. **Figure 4:**

When testing at mini-batch size 1, the runtime was slow and a RuntimeWarning was given stating an overflow was encountered. Although it was not timed properly, the entire process of training and gaining the data took well over 30 minutes to complete. In comparison to the other tests which took around 10 minutes to complete. This makes it by far the slowest.

|  |  |
| --- | --- |
| **Overall Maximum Accuracy** | **Mini-batch size** |
| 89.55% | 1 |
| 95.82% | 5 |
| 95.86% | 10 |
| 96.25% | 20 |
| 96.08% | 100 |

The mini-batch size of 20 achieves the best overall accuracy of 96.25%, and the mini-batch size of 1 achieves the worst at 89.55%.

After experimenting with different hyper-parameters. The best configuration found was mini-batch size 20 with learning rate 1.0, this achieved a maximum accuracy test of 96.45%! Epoch was set to 30.

## Verifying calculations

The output of the weights and bias can be found within the predict files attached with this assignment. Below is what was calculated versus the output:

Calculated weight: 0.09850891  
Actual weight outputted: -0.09380417

This shows there was a calculation error with the negative inverse.

Calculated weight: 0.198508917  
Actual weight outputted: 0.190532115

Calculated Bias1: 0.01491082  
Actual bias outputted: 0.03673311

Calculated Bias2: 0.01491082  
Actual bias outputted: 0.02658931  
  
Although close, this shows that there may be small errors within the calculations made in comparison to the actual output of neural network.

## Part 3 – Cross Entropy Cost function

Part 2 was using a quadratic cost function. In this part a cross entropy cost function is used. Using the same specifications, the learning rate was changed for each test as seen in figure 5.

**Figure 5:**

In comparison, to the quadratic cost there are a few noticable differences. Firstly the learning rate 100 fluctuates rather than being a constant test accuracy. The learning rate 10 is also wildly different, in the quadratic cost it starts above 80% accuracy but in entropy cost it doesn’t even reach that state even with all the tests. The learning rate of 0.001 also increases it’s accuracy much faster over the tests.

|  |  |
| --- | --- |
| **Overall Maximum Accuracy** | **Learning rate** |
| 88.27% | 0.001 |
| 96.39% | 0.1 |
| 95.97% | 10 |
| 62.81% | 1.0 |
| 10.64% | 100 |

This time the best accuracy is achieved by the learning rate of 0.1 with 96.39%.   
In the quadratic cost the learning rate 1.0 was the best accuracy at 96.45%, but is now down to 62.81% in these tests. The quadratic cost function yeilds a slightly better result (0.06%) in comparison to overall accuracy.

Figure 6 shows the accuracy test undertaken with different mini-batch sizes and with the Cross entropy cost function.

**Figure 6:**

The mini-batch size 1 took the most time to run all the tests, much greater in comparison to other run times undertaken.

|  |  |
| --- | --- |
| **Overall Maximum Accuracy** | **Mini-batch size** |
| 50.40% | 1 |
| 92.32% | 5 |
| 95.14% | 10 |
| 95.31% | 20 |
| 95.51% | 100 |

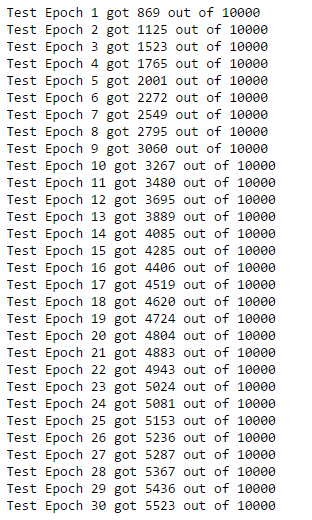
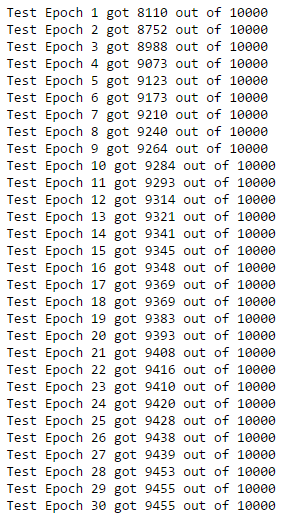
The mini-batch size of 100 achieved the best result of 95.51%, this in comparison to the quadratic cost of mini-batch size 20 which achieved 96.25% shows that there is a small decrease in overall accuracy.

During all of these tests conducted the regularization parameter was set to 0.001 the entire time. L2 regularization is used to reduce model overfitting. Eliminating overfitting leads to a model that makes better predictions, which can be seen by some of results produced. It however does this at the expense of adding bias to the estimate. Increasing the regularization parameter results in less overfitting but also greater bias. Therefore picking an appropriate estimate is essential. During testing, random subsample’s of the MINST dataset were taken a number of times and compared to get variation. This process was repeated for a slightly larger regularization parameter and the changes were observed. It was found that 0.001 produced the best results.

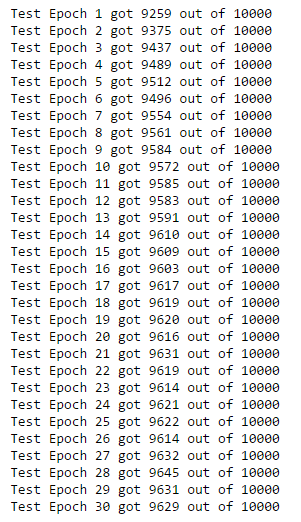
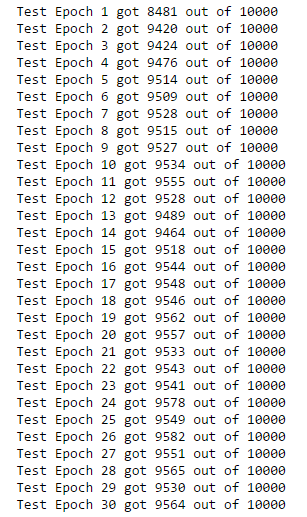
## Program output

Below are screenshots taken from the output produced by the neural network testing.

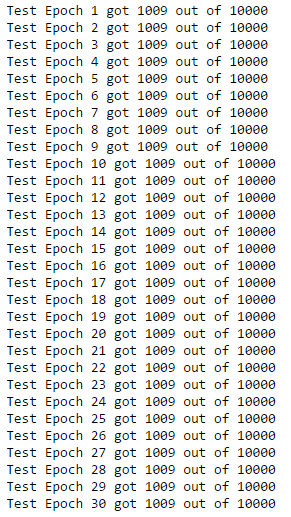
Learning rate: 0.1 Learning rate: 0.001

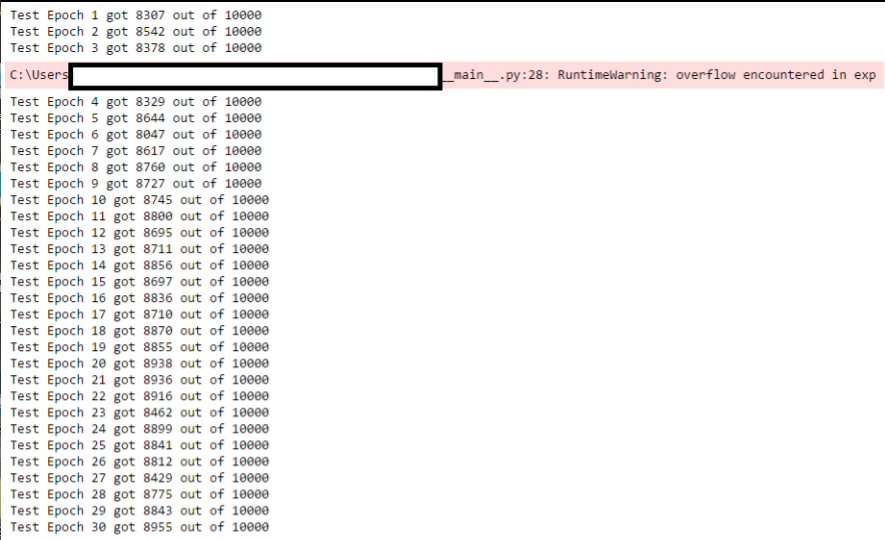
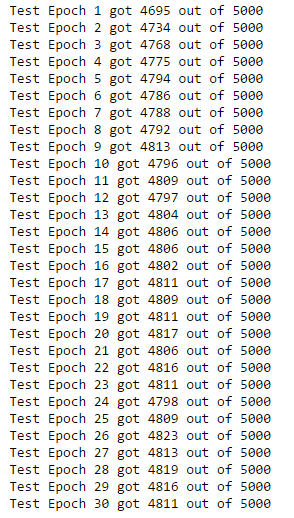


Learning Rate: 10 Learning Rate: 1.0

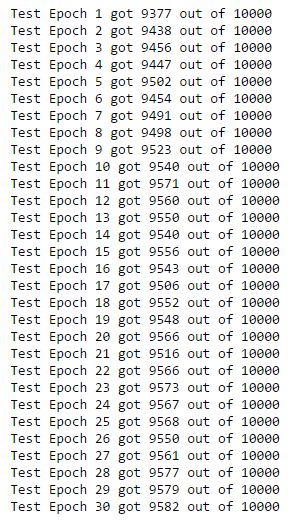
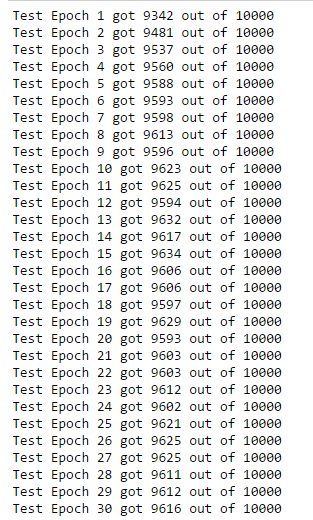


Learning Rate: 100 TestDigitX2.csv.gz at learning rate 3:

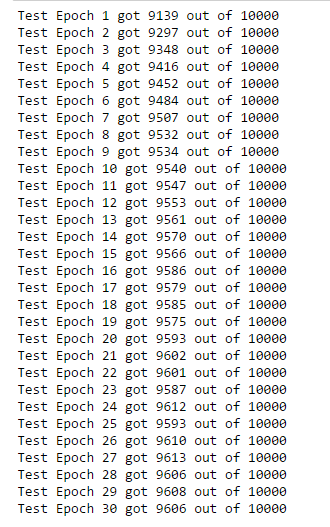
  
Mini-batch size 1:



Mini-batch size 5 Mini-batch size 20



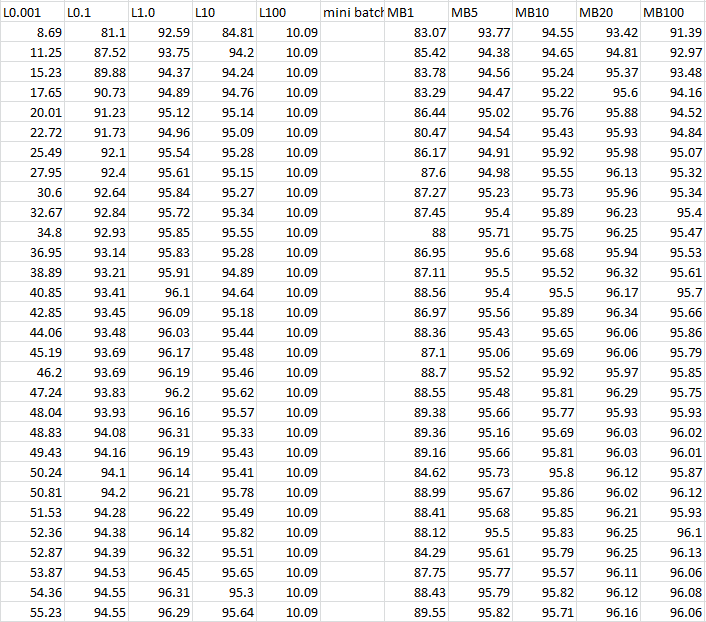
Mini-batch size 100



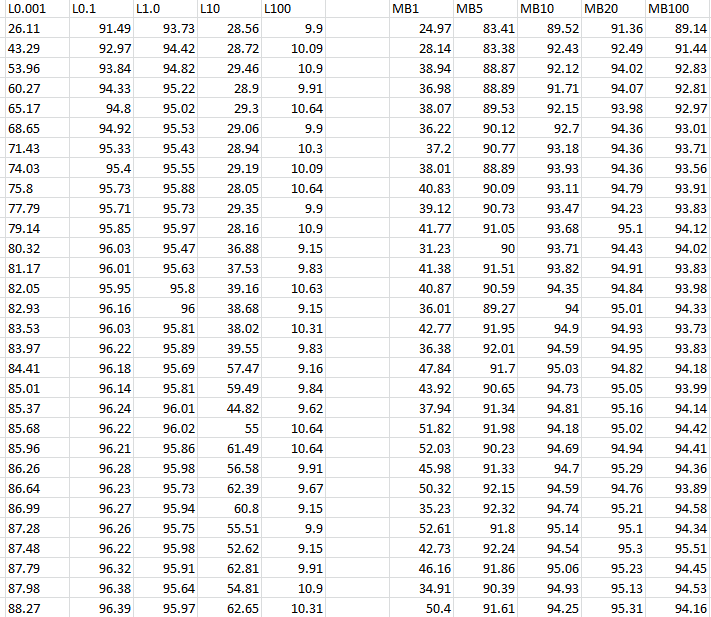
## Excel data screenshots

Below are screenshots of the data input into Excel to produce the graphs. This data was grabbed from the output of the neural network.

For the quadratic cost:



Entropy Cost:



## Sources

1. Neural networks and deep learning. 2017. [ONLINE] Available at: <http://neuralnetworksanddeeplearning.com>
2. Improve Neural Network Generalization and Avoid Overfitting. 2017.[ONLINE] Available at: <http://au.mathworks.com/help/nnet/ug/improve-neural-network-generalization-and-avoid-overfitting.html>
3. Chatbot’s Life. 2017. Regularization in deep learning [ONLINE] Available at: <https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0>
4. Neural Networks. 2017[ONLINE] Available at: <https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html>