Neural Network

Assignment 2

Carl Humphries

2018

# Part 1

## 

Figure 1: A Small Neural Network.

Given the above Neural Network, train it manually using X1 = (0.1, 0.1), X2 = (0.1, 0.2) and Y1 = (1, 0), Y2 = (0, 1). The default biases and weights are displayed in Figure 1.

## Values

## Feeding forward

First step is to pass the values through the neural network to get outputs. This is known as feeding forward, as is done as follows (Note 1’s were added as extra inputs to allow for the bias terms):

A sigmoid function is then applied to all the nets to introduce nonlinearity.

Because of the bias terms in 1’s need to be added into the to allow for the matrix multiplication. They can be considered as extra inputs or neurons coming from below.

Now using the outputs from the hidden layer neuron and the weights from the second layer, the final nets can be produced.

Applying nonlinearity to the outputs again the results are:

The final outputs can then be compared to the desired labels. This results in some measurement of error.

This will be used to compare after backpropagation to make sure the network is making progress.

## Backpropagation layer 2

Backpropagation is just finding out how much to nudge all the weights and in what direction, to minimise the error. This is known as gradient descent, because the aim is to find the gradient for all the weights and nudge them in that direction in aims to move downhill. The formula uses the chain rule to find the relations of the error to the weights:

To find these values derivations of the feedforward functions are calculated. Below is the derived functions and the matrices calculations:

## Backpropagation layer 1

To backpropagate the first layer it is slightly different but the same principles apply.

Because all of the output nodes are affected by the error and the weights of the layer before in the derivation needs to include this.

From here everything is the same just with slightly different values:

## Applying weights

Now that the gradients have been found the weights need to be adjusted accordingly. This is done using the formula below:

Subbing in what this neural network calculated and m=2 and all the errors are already sums of the samples (due to matrix calculations).

The new weights are

## Comparing error

After feeding forward with the new weights, a new error can be found:

This shows that after 1 Epoch the error was reduce by a very small amount. This may not seem like much but after many Epochs the error reduces by a lot more.

# Part 2

## Validation with Manual Calculations from Part 1

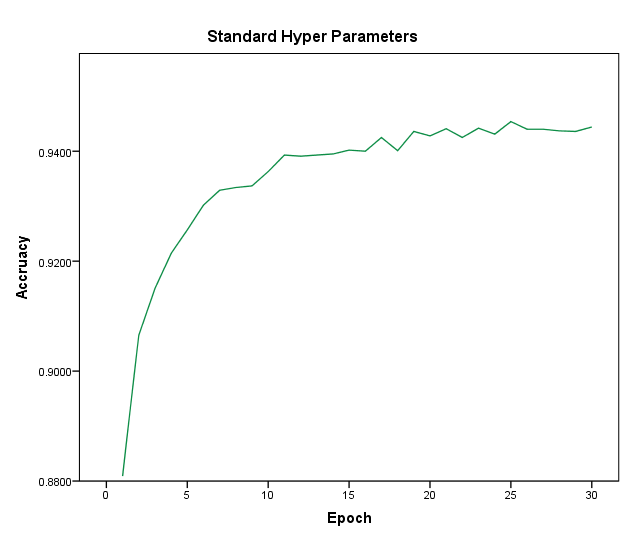
Figure 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Manually Calculated | Python Code | | |
| Weights | Epoch 1 | Epoch 1 | Epoch 2 | Epoch 3 |
| w1 | 0.0999928760822664 | 0.099992876 | 0.099985872 | 0.099978987 |
| w2 | 0.199988916763062 | 0.199988917 | 0.19997799 | 0.199967217 |
| w3 | 0.0999892155783301 | 0.099989216 | 0.099978593 | 0.099968129 |
| w4 | 0.0999985689813837 | 0.099998569 | 0.099997359 | 0.099996366 |
| w5 | 0.0993088619738889 | 0.099308862 | 0.098624152 | 0.097945815 |
| w6 | 0.0991713788595459 | 0.099171379 | 0.098350492 | 0.097537277 |
| w7 | 0.0993057052095338 | 0.099305705 | 0.098617879 | 0.097936466 |
| w8 | 0.19916741399057 | 0.199167414 | 0.198342615 | 0.19752554 |
| w9 | 0.0999287608226642 | 0.099928761 | 0.099858725 | 0.099789871 |
| w10 | 0.0998891676306169 | 0.099889168 | 0.0997799 | 0.099672175 |
| w11 | 0.0987279270749278 | 0.098727927 | 0.097467915 | 0.096219859 |
| w12 | 0.098411317559255 | 0.098411318 | 0.096837135 | 0.095277336 |

Before testing the python program on the dataset, a small test was run to simulate the Neural Network in Part 1. The results are shown in Figure 2, because they were the same after one epoch and seemed to be working well after 3 epochs it was safe to start testing on the data set. One concern was that the python wasn’t as accurate, but this was probably due to it limiting decimal places on output.

## Results for Epoch vs time with Standard Parameters

Running the network (784,30,10) with the provided parameters (η=3.0, batch size=20, epochs=30) seemed to give quite decent results (Figure 3). It seemed to fluctuate before smoothing out towards the last few epochs. This could be due to the nature of gradient decent. It also looks like it may be hitting some limit towards the last epochs. Which is probably due to the nature of the data and the limits of a neural network.

  
**Figure 3**

## Alterations in Parameters

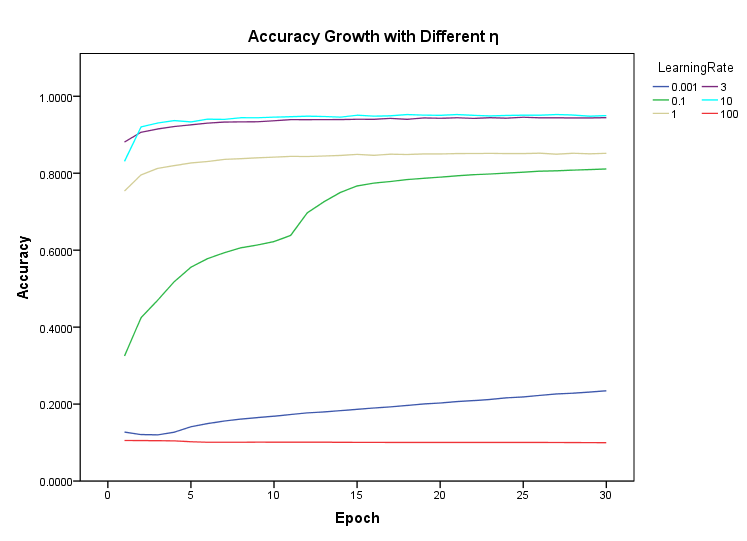
### Learning rate

Changing the learning rate seemed to have quite different and interesting results, Shown in Figure 4.

Starting with the two worst results seeming to be outliers. They both come from both ends to the testing data, 0.001 and 100. This is probably due to 0.001 learning rate, learning too slow; seen by the slight gradient in the line. Given enough epochs it most likely would be closer to the other results. Whereas learning rate of 100 jumps around too much to descend any gradient, it over shoots too often. Therefor its’ line is almost straight and sitting around 10% accurate, which is the same as guessing randomly.

There is a trend in the rest of the results showing that as the learning rate increases the quicker it jumps to higher accuracies. While this may appear good, it would intern result in hitting local maxima sooner. This is shown because the data flattens out quicker than slower learning rates.

One last thing to note is the bump in 0.1 learning rate. This may be indicating that slower learning rates can break out of local maxima’s.



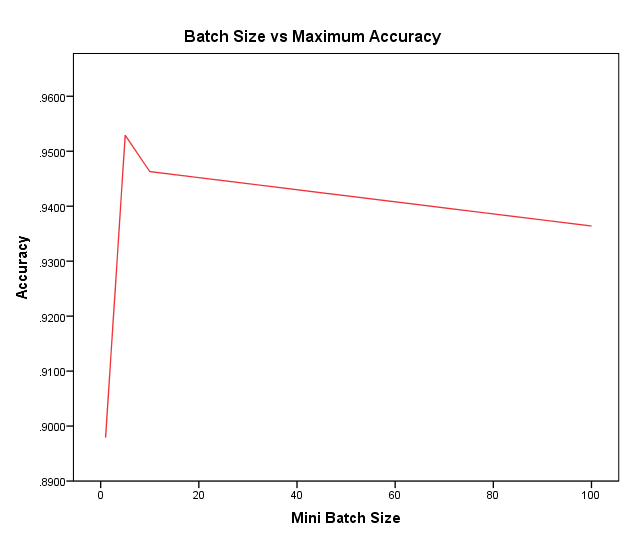
**Figure 4**

### Mini batch size

Changing how large the mini batch was affected the results in a weird pattern. Shown in Figure 5.

Starting with the most noticeable result at the peak, mini batch size 10. This could be due to smaller batch sizes than this become more bias to certain numbers resulting in poor error for different number. So, it isn’t until a size of ~10 the results seem to be bad.

As for the decrease over larger sizes, this could be explained by, the program updating the weights less. Due to less batches are calculated since the Epoch and data samples stay the same.



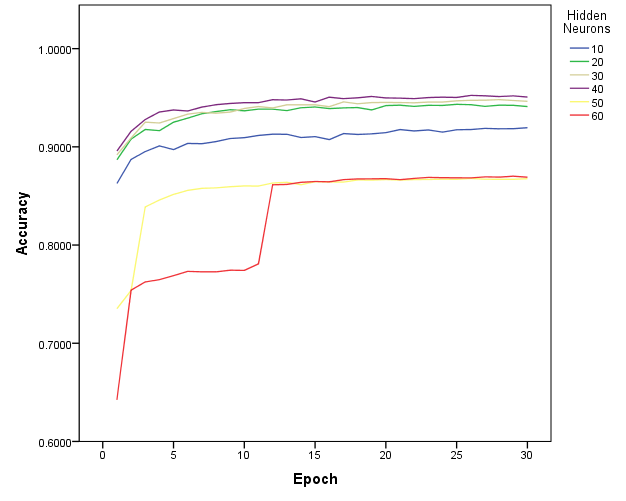
**Figure 5**

### With different Neuron size

One last thing was to change the number of hidden neurons, Shown in Figure 6. This was because all other aspects were either already tested (all parameters) or couldn’t be changed within the scope it this neural network (different input neuron size or output neuron size). More Epochs could of be allowed but it’s quite predictable what would happen after 30.

Starting with higher neuron count it is quite apparent they do not perform better. This could be due to large activation and too much for the network to scale and find equilibrium. But the results do show that more neurons lead to better results when <50.

A different approach could be to increase the layers rather than the neurons alone.



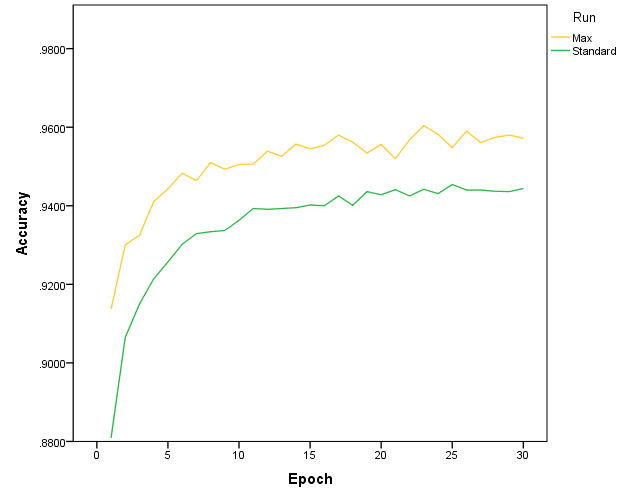
**Figure 6**

### Experimentation

After trying multiple combinations of hyperparameters (Trying the best from pervious tests). The best results were from:

* Hidden neurons = 40
* Learning rate = 5.0
* Mini batch size = 10

The results are compared to the standard test in Figure 3, shown in Figure 7. It is almost like it has been shifted up but with more fluctuations. All the testing was done on 30 Epochs, but given more Epochs the results most likely would have been better.



**Figure 7**

# Part 3

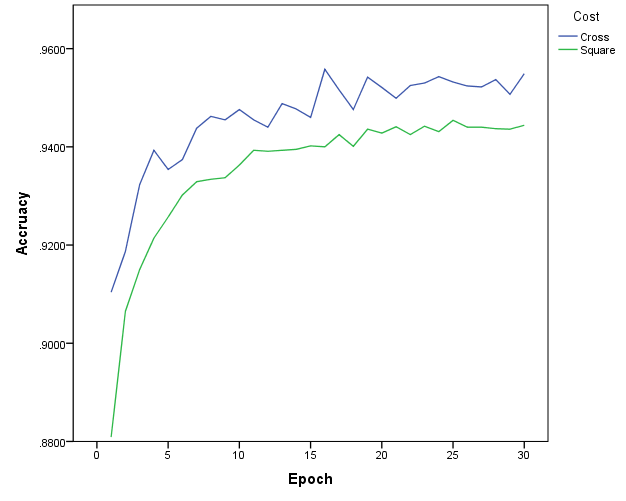
### Cross Entropy cost function

Throughout Part 2 and Part 1 the Neural Networks used the sum of squares cost function shown in Part 1. One final test was to change the cost function to the Cross Entropy given by:

Because of the nature of using the Cross-Entropy cost function, the Neural Network needs to have a different activation function on the output layer. Because of this SoftMax is use in place of sigmoid to allow for a probability distribution.

This doesn’t change too much of the code since the derivation for the SoftMax is the same as the sigmoid. But the second layer derived relations end up cancelling out to make a simpler function:

After applying this to the code, it gives the following results, shown in Figure 8. The cross-entropy cost function seems to be generally better but with more chaotic fluctuation. It still follows the same general curve just with slight raise.



**Figure 8**

# Conclusion

The neural network proved to be valid in Part 1 and a lot faster programmatically than doing it manually. Although the error only changed slightly it was enough to say it was doing something positive.

Parameters have sometimes subtle effects on the outcome of the accuracy but if they are tweaked just right it can be the difference in 1% which is a lot when dealing with large amounts of data.

Thought testing the sum of squares error was enough to achieve great results. But the Cross entropy seemed to do better just in a more chaotic fashion.