Intelligent Systems

ASSIGNMENT 2 REPORT PART 3

Himath Ratnayake
S5209861 | GRIFFITH UNIVERSITY 2021

Software Design

Functions:

The main data structures that are used for a neural network are contained within a python class. The output of a neural network is affected by the values of the weights and biases, and so when training the neural network the main idea is to tune these weights and biases such that the prediction strength increases. For each iteration of the training process, known as an epoch, the following main steps are done:

1. The predicted output is calculated with feedforward process
2. The weights and biases are updated with backpropagation

Neural network class (NN)	
Description	This is the main class through which the neural network and its layers, neurons, and associated inputs are initialized. It contains all the functions necessary for a neural network to work.
Data Types/Structures	**Constructor:** Within the NeuralNetwork class constructor, the input parameters entered into the program are initialized. By default, the NInputs are 784 for the input layer, NHidden is 30 and NOutput is 10 (each number pertains to the number of neurons). The epochs, learn rate, and batch size are also initialized. Finally, with the initial construction of the object, random weights and biases must be given to all neurons in the hidden and output layers, which is done using the np.random.randn function, which will assign a random number from a normal distribution between 0 and 1 to each of the neurons in the hidden/output layer.

train	
Description	This is the main driver function within the Neural Network class that trains the neurons.
Data Types/Structures	First, a for loop is created, which will iterate from 1 to the number of epochs + 1, with each epoch representing one cycle in the data where the values for each neuron were slightly tweaked and all mini batches were processed. Next, the training data is randomly shuffled using random.shuffle, and a "batches" list is initialized. Then, in another for loop, the training data which has now been shuffled is divided up by index into minibatches of the size specified by the user. Each minibatch is added into the parent "batches" list. The use of mini batches with random sets of data points is what makes this neural network stochastic. Next, each of the minibatches in the batches list is processed using the process function. After the batch processing, evaluation is done on the test data with the tuned weights and biases found for the current mini batch. For each example in the test data, the output_neuron function is called to find the output neuron with the highest activation using the feedforward method. If the output neuron with the largest amount of activation returned from the feedforward function is the same as the test point's label value, then a variable "correct" is incremented by 1. Finally, once all test data has been analyzed, the number of correct items for this epoch is returned to the train function to be shown.

sigmoid	
Description	Activations must be a specific value between 0 and 1, and for this, a sigmoid function is used; in this function, very negative inputs are closer to 0 and more positive inputs are closer to 1. Hence this activation of the neuron is a measure of how positive the neuron is.
Data Types/Structures	For the sigmoid function, a value "x" is passed in, and then substituted into the following equation. The answer is then returned: $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

output_neuron	
Description	This function is used to return the updated values for the neurons in the output layer, in a feed forward process.
Data Types/Structures	The sigmoid function is called on the sum of dot product matrix of the weights and biases for the hidden layer. Essentially, the variable "perceptron" is the answer to the dot product between the input features and weights, with the bias then added on top. The sigmoid then takes this perceptron and applies the sigmoid logistic function to it, outputting the layers the hidden layer neurons. Next, another output perceptron is found, which is equal to the dot product of the output weights and hidden layer neurons found, with the output biases then added on top. This output perceptron is once again passed into the sigmoid function, which will return the output layer neurons. Of the output layer neurons, the one with the highest activation is found using the np.argmax function and then returned.

evaluate_batch	
Description	This function is responsible for processing each mini batch. Here, each data point in the mini batch is run through the backpropagation function to find errors for all the weights/biases. Note that all the lists with neurons are numpy arrays meaning arithmetic done on such a list will apply to all elements inside said list.
Data Types/Structures	In this function, an error bias and error weight list is initialized with zeros using np.zero for each weight and bias, based on the shape of their data. Then, for each image's data and label in the mini batch, backpropagation is used to find new error and new bias values for both the hidden and output layers. The data values for each pixel are also divided by 255, such that all values fall between 0 and 1. After sending the data and label for the given point in the mini batch into the back propagation function, a set of new bias and weight errors are returned from the BP function for the hidden and output layers. Then, the previous initialized bias and weight error lists are updated, with the new error values found and stored in the new_biaserrors/new_weighterrors lists being added to each corresponding index of the original bias/weight error lists for both the hidden and output layers. Then, for each of weight in the class itself, the updated values from weight_errors are added. When doing this, each of the values in weight_errors is multiplied by a "factor" based on learning rate initialized in the main function. Doing this determines the "step" (i.e. the magnitude) of change that each change has on the data. The same process is done with the biases in the hidden and output layers.

back_propagation	
Description	Backpropagation is used to optimise the weights so that the network can improve the way it maps inputs to outputs.

Data Types/Structures	The back propagation begins with a feed forward, which is done in the same way as explained above in the output_neuron function.
	Then, with backpropagation, the weights in the network are updated so that the actual output can become closer to the intended target output by minimising error for each output neuron.
	$$\frac{\partial C}{\partial w_L} = \frac{\partial i_L}{\partial w_L}\frac{\partial o_{L1}}{\partial i_L}\frac{\partial C}{\partial o_{L1}}$$
	First, we must calculate the change in error for each output neuron. I.e. "how much does the input value to the neuron change when the weight changes?" The input to the neuron's derivative with respect to the weights in the last layer is the output from the previous layer. $\frac{\partial i_L}{\partial w_L}$
	Given an output value from the neuron, we must also then find out how much of the output from the neuron changes when the input changes. The function that transforms an input to the output is the sigmoid activation function. Hence, for this, the slope (derivative) of this function must then be found using the deriv_sigmoid function. The derivative is calculated through chain rule as follows:
	derivative = outputneuron * (1.0 – outputneuron) $\frac{\partial o_{L1}}{\partial i_L}$
	Finally, we must figure out how much the cost changes when the output from the neuron changes – this is the derivative of the cost function (e.g. quadratic cost), with respect to an output value. $\frac{\partial C}{\partial o_{L1}}$
	This can then be used to update the biases for the output error, in a similar process to how the weights will be updated. Bias is a parameter in the network used to adjust the output along with the weighted sum of the inputs to the neuron must also be accounted for.
	Next, for the hidden layer, a similar process is repeated for all weights from the hidden layer to the output layer, with the main difference being we use data we've already calculated from the output layer instead of another derivative of cost function. For each neuron in the hidden layer, its weights are used to calculate the error based on the slope of the hidden layer neurons and the output values found before. This connection is calculated with matrix multiplication.
	`hiddenlayer_derivative = hiddenlayer_neurons * (1 - hiddenlayer_neurons)`
	`error = np.dot(weights, output_error_bias) * hiddenlayer_derivative[neuron]`
	Then, once the hidden layer's error weights are found using the bias error, the amount of error in the hidden and output layers for both bias and weights are returned back to the evaluate batch function, so that they can be updated with these new values.

quadratic_cost	
Description	This function returns the output for the derivative of quadratic cost, that is used in backwards propagation
Data Types/Structures	$$C(w,b) = \frac{1}{2n}\sum_{i=1}^{n} \| \| f(x_i) - y_i \| \|^2$$ From the original quadratic cost function, the derivative in respect to $f(x_i)$ simplifies to $f(x_i) - y_i$ (See Appendix 1).. Essentially, a numpy array of 0s is initialized for the test label vector y_i., and the current label's number is set within this array of 0s as "1". After this, the test instance vector $f(x_i)$ (the output layer neurons passed in), is subtracted from the test label vector, and then returned to the main function: *test instance vector – test label vector* As both are numpy arrays, this subtraction is made simple as the contents of the arrays are matched by index too.

cross_entropy	
Description	This function returns the output for the derivative of the cross entropy cost, that is used in backwards propagation as an alternate function.
Data Types/Structures	$$C(w,b) = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln\left[f(x_i)\right] + (1-y_i)\ln[1 - f(x_i)]$$ The above equation can be derived to get a final equation as below: (see appendix 2 for derivation) $$= -\frac{t_j^{\grave{x}}}{o_j^x} + \frac{1 - t_j^x}{1 - o_j^x}$$ **Where:** • T^x_j = test label vector • O^x_j = test instance vector Similar to the quadratic cost function, it is a matter of substituting the values where appropriate in the function, and then returning it to the back propagation function for use.

Results

Part 1: Testing With Default Hyperparameters

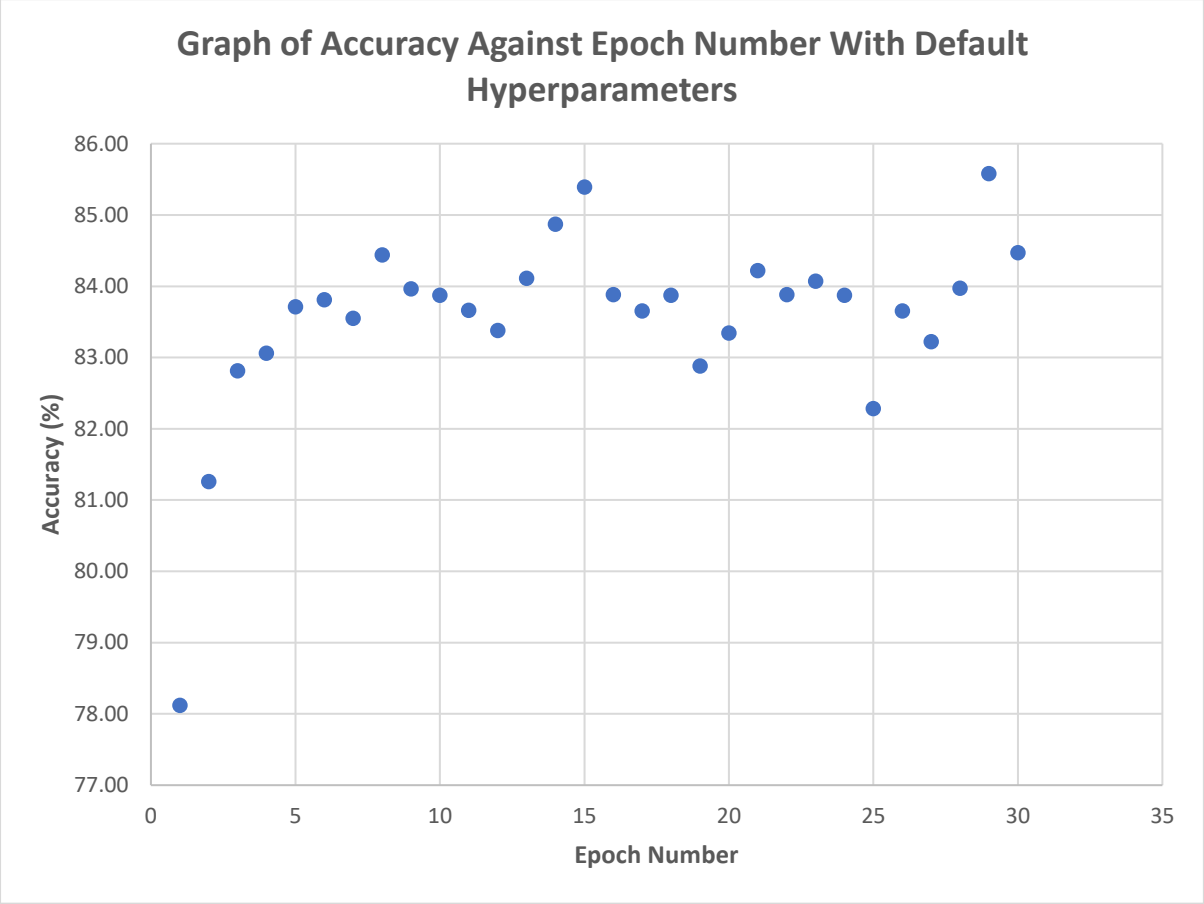

Figure 1: Testing set accuracy vs epoch number for the neural network with default parameters

Within 30 epochs with mini batch size of 20 and learning rate of 3, the neural network was able to achieve an accuracy of **85.58%.** Within the neural network, the increase in learning was not consistently upwards. Although the accuracy was generally increasing with each epoch, after a certain number of epochs it started to fluctuate up and down slightly. Logically, this could be expected; one reason possibly being due to the learning rate. The learning rate controls how quickly a model can adapt to the data set, and larger learning rates can lead to more rapid changes or fluctuations, but with the advantage that less epochs are needed to reach a high accuracy – note that in this training set for example, over 80% accuracy could be achieved by the second epoch as seen by figure 1. Another reason could be due to overfitting – in the first epoch, the accuracy is the lowest due to underfitting, as the weights and biases haven't been tuned well enough in just that one epoch to accurately predict as well as it could. However, after around the 16th epoch, overfitting can come into play, where the model becomes too reliant on the training set data, and so it's ability to predict for yet unseen data is hindered. Rather than learning from the data it starts memorising the training data itself which is not ideal.

Part 2: Exploring Different Learning Rates

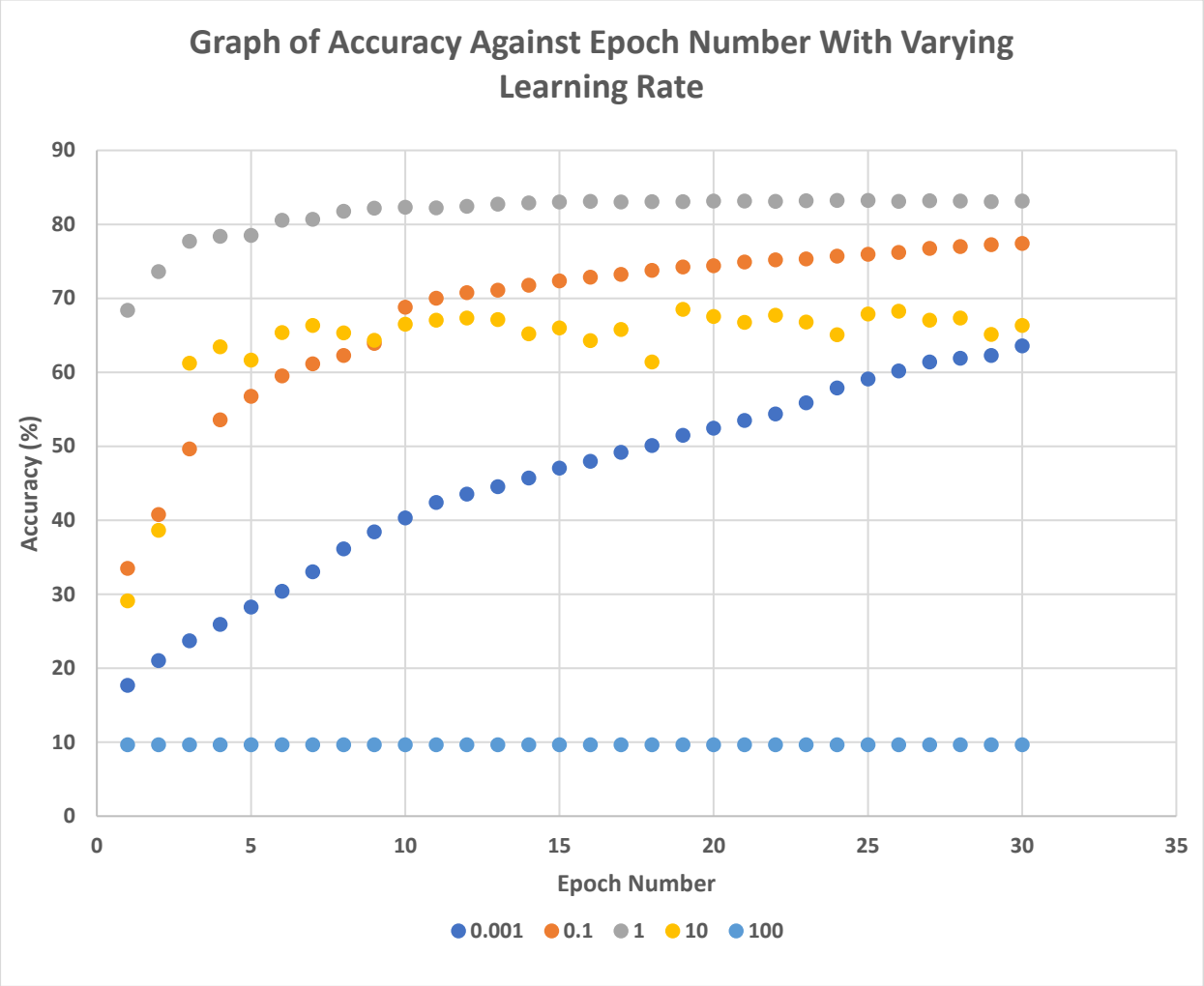

Figure 2: Graph of neural network model accuracy vs epoch with different learning rates

Table 1: Highest report accuracy with various learning rates					
Learning Rate	**0.001**	**0.1**	**1**	**10**	**100**
Highest Accuracy Output (%)	63.6	77.44	83.26	68.52	9.66

The learning rate of the neural network, also known as the step size, can be described as the magnitude of the amount each step makes in the direction of a minimum in the data.

As can be seen by figure 2, each learning rate presented a noticeable different progression of accuracy increasing. With a learning rate of 0.001, it was found that learning was smoothest – each "step" in learning with an epoch was very small, but that accuracy increase was consistent. With a learning rate as low as 0.001, much smaller changes are made to the weights of the neural network with each update. Because of this, more training epochs are required to arrive at a more accurate or optimal solution. Hence, with only 30 epochs the learning rate of 0.001 could only find a highest accuracy of 63.6%, but with a higher number of epochs, this accuracy value has the potential to become higher.

In contrast, at 0.1 learning rate, the accuracy increase was still quite consistent, but there was also the added bonus that it was able to find a more accurate solution faster, with it having a highest accuracy found of 77.44% as seen in figure 2.

The learning rate of 1 was able to find the highest accuracy of all learning rates tested, with an accuracy score of 83.26%. By the first epoch, learning rate of 1 was able to produce an accuracy on the training set of 68.42%, which was already higher than the maximum accuracies found for learning rates of 0.001, 10 and 100.

Because the step size is lower when learning rate = 0.001, 0.1, and 1, the results didn't fluctuate in accuracy as much when predicting for the testing set, as overfitting takes a longer number of epochs to happen when the step size (learning rate) is lower. However, the results of overfitting after a certain number of epochs has passed has the potential to be more prevalent with lower learning rate too, as the model will start basing itself too much on the training data. Furthermore, underfitting can take place for a longer number of epochs too with lower learning rate, and so it takes longer to reach a complex enough model to consistently get high testing accuracy. Thus, getting a balance of learning rate such that overfitting and underfitting are minimised is crucial.

The learning rate of 10 started off with a lower accuracy than learning rate 0.1 and 1 in the first epoch, and then reached a maximum accuracy of 68.52 at the 19^{th} epoch. As can be seen by the graph, learning rate of 10's curve is much more unstable, with much greater fluctuation in results being visibly shown in contrast to the smoother curves for lower learning rates. This lower accuracy with increased training rate can potentially be due to underfitting, as the model isn't able to completely capture the data's information due to how large the step size is, and so the model is not complex enough.

Finally, the learning rate of 100 could only find a single accuracy over 30 epochs, which was 9.66%. This was the lowest of accuracy exhibited by far when comparing to all other learning rates tested. This is because at a learning rate of 100, the model is significantly more prone to situations such as ones where it can ignore and overshoot a local minimum. Moreover, it can also cause the performance to diverge or "get stuck", as the loss function is best when approximating locally. Hence, when the learning rate is too high and huge steps are being made, it's possible to enter a completely different parameter space from the previous point that the network was in, which will throw off any progress towards a more accurate classification model being made.

Thus, of the learning rates tested, learning rate of 1 was found to be the most effective, as it is not too large as to make the model converge too quickly at a suboptimal solution, and not too small that it takes a large amount of epochs to arrive at an acceptable accuracy.

Part 3: Maximum Accuracies With Varying Mini Batch Size

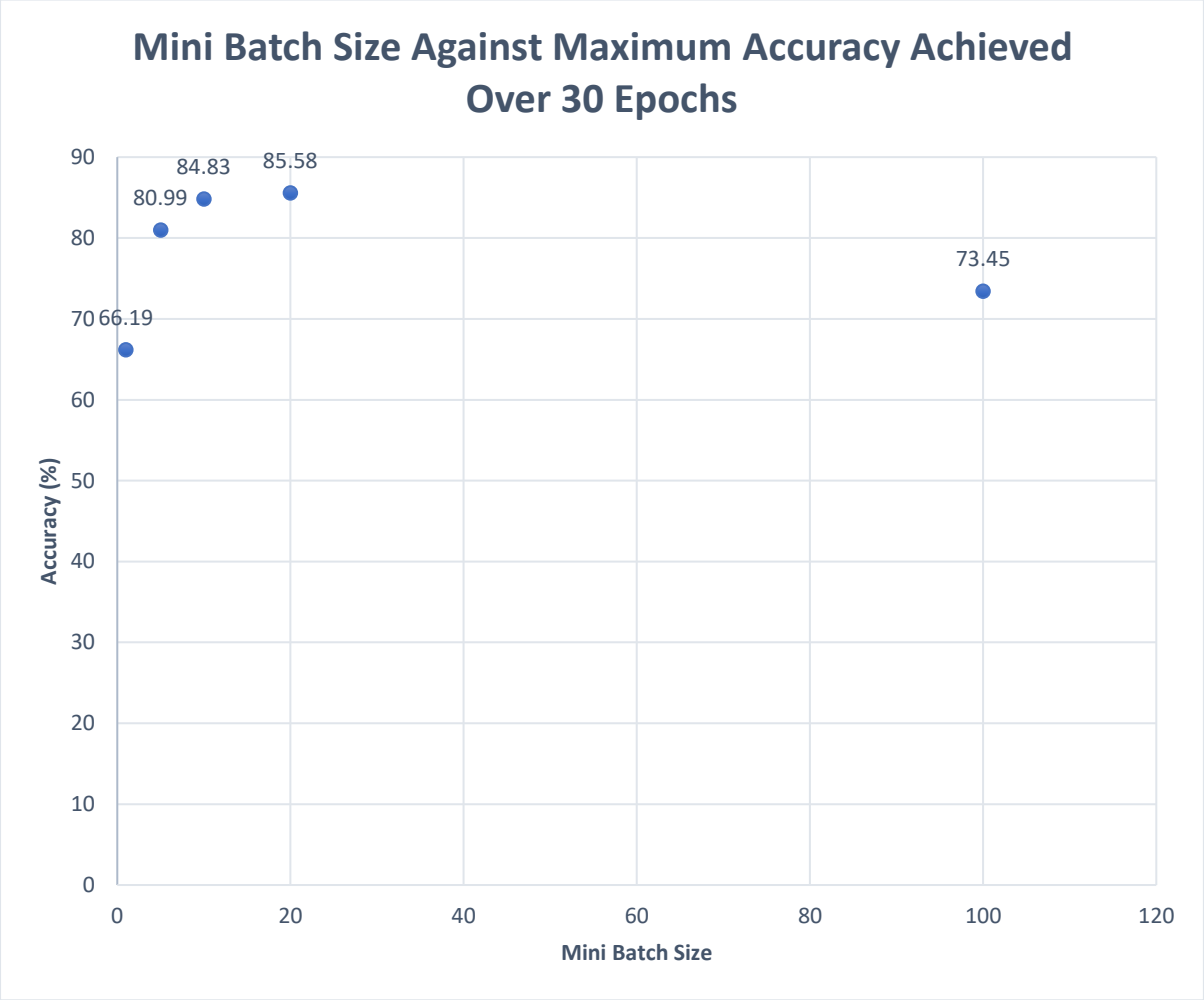

Figure 3: Graph of neural network model maximum accuracy vs epoch with different mini batch sizes

Table 1: Average time taken per epoch over 5 trials					
Batch Size	**1**	**5**	**10**	**20**	**100**
Time Taken (seconds)	147	121	112	116	119

A mini batch size of 1 is equivalent to a stochastic gradient descent where each individual training example is analysed and the model is updated for every single training point. The frequency of these updates means that the model training is more computationally expensive, which can be seen from the fact that it took the longest average time per epoch at 147 seconds as noted in table 1. It also seemed to have a negative impact on training accuracy, as batch size of 1 showed the lowest accuracy, with the maximum accuracy achieved for this batch size being 66.19% over 30 epochs. When testing, there also seemed to be a greater fluctuation and variance in accuracy over the epochs, which could allude to a particularly “noisy” gradient, with noise being aspects of the data that do not add additional useful information to the training of the network; the smaller the mini batch is, the noisier the gradient. One countermeasure to alleviate this could be to decrease the learning rate so that stability can be increased.

With mini batches greater than 1 being used, it was found that both computation speed per epoch and maximum accuracy increased for mini batch size 5, 10, and 20. Mini batch size 10 was found to be the fastest computationally at 112 seconds per epoch, and achieved a maximum accuracy of 84.48%, which is comparable to the 85.58% achieved with mini batch size 20, which took slightly longer at 116 seconds average per epoch, but was also more accurate.

The mini batch size of 100 however exhibited noticeably lower accuracy when compared with batch size 5, 10, and 20, but the accuracy was not as low as mini batch size of 1. The reason for this could be related to the concept of gradient signal noise elaborated upon earlier. Noise allows the algorithm to have a chance at jumping out of a bad local minimum and allow it to find a potentially more accurate minimum, or ideally, the global minimum. Here, what may have happened is that due to the lack of noise due to such a large batch size of 100, the network simply converged at a more suboptimal minimum. Over the epochs, the average time taken for mini batch size of 100 was also found to be 123 seconds, which is an increase in average time per epoch from both batch size 10 and 20.

Part 4: Finding Ideal Hyper Parameter Settings

Along with the settings explored above, a variety of other hyper parameter settings were tested. Of them, the maximum accuracy found was **89.98%.** This was achieved after **20 epochs**, **learning rate of 4** and **mini batch size of 15**. It was found that after 20 epochs, the accuracy seemed to plateau, and thus computation time could be saved by reducing the epoch number from 30 to 20. On top of this, beyond 20 epochs overfitting's effects became more obvious and had a chance to hindering overall accuracy when encountering unseen training data. At mini batch size of 15, accuracy also improved when compared to the default parameters due to a sufficient amount of gradient noise still being able to be introduced; this would allow the network to have a better chance at escaping suboptimal local minimums they may encounter. Finally, the learning rate being slightly increased to 4 was able to result in a better accuracy in a lower amount of epochs, while also minimising large fluctuations in accuracy which could be caused by a higher learning rate. When the learning rate was higher than 4 but less than 10, the network was able to find an accuracy greater than 80% the fastest – usually within 5-10 epochs at most – but plateaued soon after with some minor fluctuations in accuracy.

Part 5: Cost Functions

A variety of hyper parameters were tested for the cross entropy cost implementation. The cross entropy cost is essentially the idea of how surprised on average an individual is at learning the true value for a neuron. This makes the cross entropy cost particularly good when predictions differ greatly from the actual learning. Cross entropy cost is often preferred for classification problems whereas quadratic cost is more preferred for regression problems.

Table 2: Hyperparameters tested for Cross Entropy Cost		
Mini Batch Size	**Learning Rate**	**Maximum Accuracy (%)**
20	3	84.89
15	4	83.84
25	0.1	10.14
25	1	74.24
25	10	42.27
1	1	49.49

Of the hyperparameters tested, none were able to match the maximum accuracy of quadratic cost of 89.98%. With the same ideal hyperparameters of batch size = 15 and learning rate = 4, the cross entropy cost function was able to reach a maximum accuracy of 83.84%. Notably, at low learning rates the cross entropy cost function's accuracy became considerably lower, as demonstrated by the learning rate = 0.1 point only reaching a maximum accuracy of 10.14%. The cross entropy cost function was also more volatile when it came to predicting accuracies, with the accuracies found between epochs tending to have much greater fluctuation than with the quadratic cost function.

Conclusions

Through the implementation of this neural network, the highest accuracy found was 89.98%. This was achieved with hyper parameter settings where the mini batch size was 15, epoch number was 20, and learning rate was 4. With these settings, a balance could be found where mini batch size was high enough that the data could be processed in a relatively efficient amount of time while also preserving accuracy through added gradient noise. Furthermore, the learning rate was balanced such that it wasn't large enough to make the model converge too quickly at a suboptimal solution, and not small enough that the amount of epochs taken to reach an acceptable accuracy is too large. The number of epochs was also reduced such that overfitting on the training set was minimised. In testing with various mini batch sizes, batch size of 20 yielded the highest accuracy while batch size of 1 yielded the lowest. A learning rate of 1 was found to maximise accuracy out of the given learning rates to test, while a learning rate of 100 made the solution converge prematurely at a very suboptimal model.

Of the cost functions quadratic cost was found to produce a greater accuracy value more consistently, with cross entropy cost being more prone to fluctuating accuracy with greater training rates. For the default parameters tested, it was able to produce a maximum accuracy of 84.89% over 30 epochs.

Appendix 1: Derivative of Quadratic Cost Function

```
f(a) = 1/2 (a-y)^2

df / da = d[ 1/2 (a-y)^2 ] / da
        = 1/2 d[ (a^2 -2ay + y^2) ] / da
        = 1/2 ( d[ a^2 ] / da - 2 d[ ay ] / da + d[ y^2 ] / da)
        = 1/2 ( 2a - 2y + 0 )
        = a - y
```

Where:

- A = test instance vector
- Y = test label vector

(Source: https://stackoverflow.com/questions/37688536/how-to-correctly-derivate-quadratic-cost-function)

Appendix 2: Derivative of Cross Entropy Cost Function

$$
\begin{aligned}
\frac{\partial E_x}{\partial o_j^x} &= \frac{\partial}{\partial o_j^x}\left(-\sum_k [t_k^x \log(o_k^x)] + (1-t_k^x)\log(1-o_k^x)]\right) \\
&= -\frac{\partial}{\partial o_j^x}\left(\sum_k [t_k^x \log(o_k^x)] + (1-t_k^x)\log(1-o_k^x)]\right) \\
&= -\frac{\partial}{\partial o_j^x}\left([t_j^x \log(o_j^x)] + (1-t_j^x)\log(1-o_j^x)]\right) \\
&= -\left(\frac{t_j^x}{o_j^x} - \frac{1-t_j^x}{1-o_j^x}\right), \text{Chain rule} \\
&= -\frac{t_j^x}{o_j^x} + \frac{1-t_j^x}{1-o_j^x}
\end{aligned}
$$

Where:

- T^x_j = test label vector
- O^x_j = test instance vector

(Source: https://stats.stackexchange.com/questions/370723/how-to-calculate-the-derivative-of-crossentropy-error-function)

References

Numpy Neural Network References (used as a guide to get an understanding for backpropagation equations and numpy frameworks but none of the code from these websites was directly used; the AI problems and data sets they dealt with were not relevant or particularly similar to this fashion item dataset but effectively demonstrated the use of numpy to make the neural network more efficient, as well as techniques for storing weights/biases in numpy arrays for easy arithmetic on all neurons).

https://towardsdatascience.com/lets-code-a-neural-network-in-plain-numpy-ae7e74410795

https://www.kdnuggets.com/2019/08/numpy-neural-networks-computational-graphs.html

https://realpython.com/python-ai-neural-network/

https://medium.com/analytics-vidhya/step-by-step-building-a-neural-network-from-scratch-using-numpy-only-build-a-sentiment-b76393417291