Project 3 - MNIST Digit Recognition Using Multi-Layer Neural Networks

Adrian Cross

November 25, 2019

across19@vols.utk.edu
Machine learning
COSC 522

1 Abstract

In this report the evaluation of a neural network, Nielsen [2015], has been performed. The neural network structure has been outlined by investigating the technical details. Using this, hyperparameters have been defined and then varied in an evaluation, graphing the accuracy and cost values at each epoch of the neural network. From these graphs an approximate optimum values can be seen for each hyperparameter, depending upon the accuracy value required by the analysis being performed.

2 Introduction

2.1 Structure

Machine learning can be a powerful tool in the recognition of handwritten digits. By utilizing a 'neural network' handwritten digits can be input, processed and output into the computer as symbols ready for a range of different processes. For example a handwritten log of sales can be input into a computer in order to digitize its contents via a neural network. In Nielsen [2015] a neural network has been developed which performs such an operation, with varying degrees of accuracy.
This network is made up of multiple 'neurons'. Each neuron takes one or several inputs, a weight depending on the inputs and then outputs based on these weights and inputs. Section 3.3 shows more detail on how different specific neurons can work. These neurons are then layered side by side in layers as shown in figure 1.

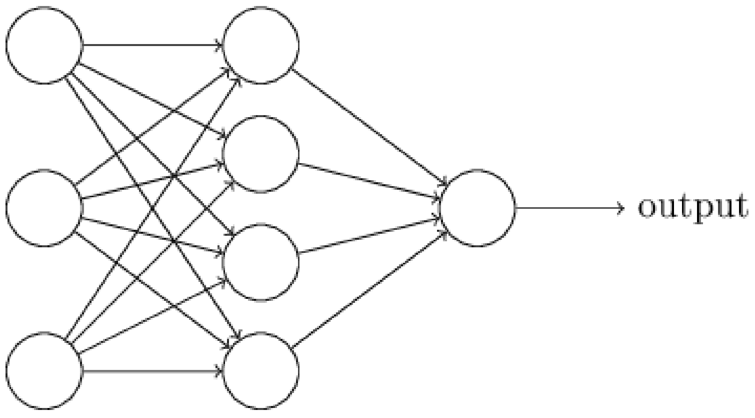

Figure 1: Small neural network with three layers made up of an input, output and hidden layer [Nielsen, 2015].

The first layer of this network is called the 'input' layer which is where the initial data is inputted. The last layer is the 'output' layer where the final, digitized output is outputted to. The middle layers is where the most processing occurs, in these layer(s) the data is input into the neurons, processed, and then output into the next layer. The more layers used in the neural network, the higher the computational cost however it can also increase the accuracy of the neural network. The number of layers is an example of a variable hyperparameter, which is described in section 2.2. After data has passed through the network once, called an epoch, it can then be put through again with the updated weights for each neuron, allowing for a more efficient calculation after each epoch has passed.

2.2 Hyperparameters

The accuracy of the network evaluated depends on the written code itself as well as a variety of inputs into the code, called hyperparameters. In addition to the previously described number of hidden layers hyperparameter, in this evaluation the learning rate and minibatch value are also investigated. The mini batch size is where the input training data is split up into smaller batches for efficiency, then, when the network is trained, it is trained on small batches of data instead of over the whole data set. Therefore a high mini batch size will decrease computational efficiency, but will increase accuracy and vice versa. The training rate is the incremental step size taken when training the data set. A large incremental step size will increase computational efficiency, but can cause the program to skip over certain parts of the data, such as a minima when used in gradient descent, see section 3.2. In this report the structure of the network is described and then the hyperparameters varied, to see how the accuracy is altered.

3 Technical approach

3.1 Batch vs online processing

It is important to form a distinction between batch and online processing for the purposes of the machine learning techniques used in this code. Online processing can be thought of as a more adaptable way of machine learning as it is constantly updating the weights for each input sample. As a sample is processed during online processing its weight is updated, therefore obtaining a set of weights which is different for each input sample. Batch processing keeps these system weights constant, but computes an error associated with each input sample and updates with each epoch, as opposed to updating each sample. This generally makes the batch algorithm slightly more computationally efficient[Zheng et al., 2017]. Figure 2 shows a workflow diagram for these two different learning techniques.

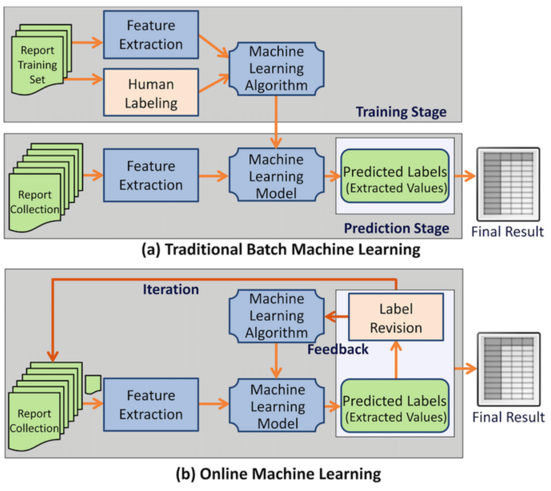

Figure 2: Online machine learning vs batch learning workflow diagrams. (a) is batch machine learning and (b) is online machine learning [Zheng et al., 2017]

3.2 Batch gradient descent vs stochastic gradient descent

An example of the difference between batch and online processing is batch gradient descent compared with stochastic gradient descent, where stochastic gradient descent is an example of online processing. Both these techniques are used to find minimas or maximas however different methods have different advantages in computing power and accuracy, depending on input parameters such as the number of data points.

In batch gradient descent the gradient is calculated from the whole training set, updating the weights via equation 1.

$$\Delta W = \eta \sum_{i=1}^{N} (y^i - \phi(w^T x)^i) x^i \tag{1}$$

Where η is the learning rate, ϕ is the activation function and N is the total number of data points [scikit]. Figure 3 Shows how gradient descent can be used to find a local minima.

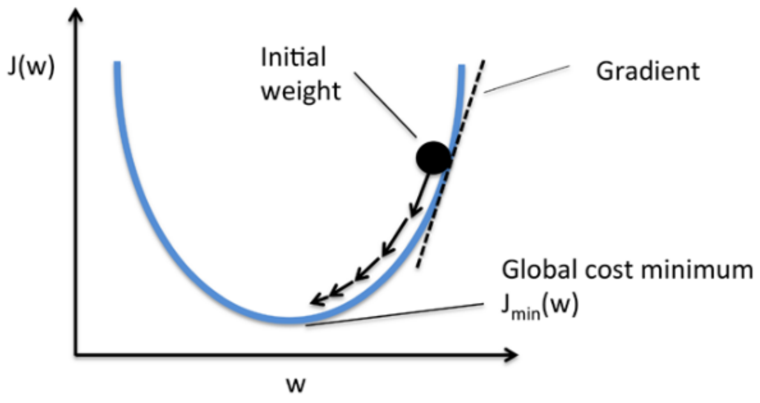

Figure 3: Demonstration of gradient descent [Raschka, 2015]

In batch gradient descent the gradient is calculated from the whole training set, indicated by the sum in equation 1. If a large dataset is used, this makes finding the minima very computationally intensive as you have to loop over large number of data points and reevaluate the training dataset during each step.
In stochastic gradient descent a similar method is used, however the weights are now updated based on the accumulated error over the samples. Using this the weights can be incrementally updated with a single training sample instead of the whole training set. This allows for a quick update for each step with one data point instead of the whole data set, making it much less computationally intensive for large number of data points. This process can be seen in steps outlined below[scikit].

1. Choose an initial vector of parameters w and learning rate η
2. Repeat until an approximate minimum is obtained
3. randomly shuffle examples in the training set
4. for i=1,2,... , N do $w = w + \Delta w$

For the neural network evaluated in this report, a stochastic gradient descent learning algorithm is used [Nielsen, 2015].

3.3 Perceptron vs sigmoid neurons

Perceptron is a way of structuring neural networks. A single layer perceptron is done by taking several binary inputs, with weights, and produces a single binary output. This process of a single layered perceptron is shown in figure 4.

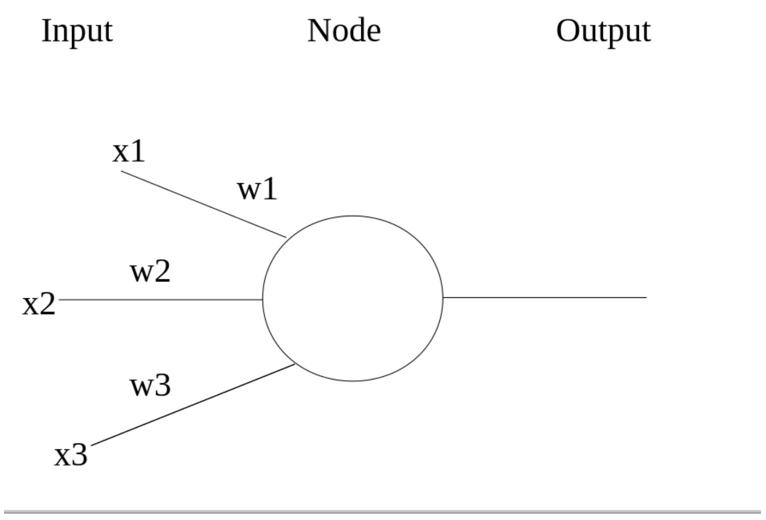

Figure 4: Single layered perceptron

This process can be layered on top of each other to produce a neural network of multiple layered perceptron with the output of one perceptron, which will be 0 or 1 depending on the threshold value, being the one of the inputs of the next layer. A bias can also be included to alter this threshold value. Depending on the bias value it will either make a perceptron node easier or harder to output a 1. The weights and bias for perceptrons can now be automatically tuned using machine learning. This requires the use of sigmoid gates as the use of perceptron neurons can vastly alter the output for only a small change of the input.
This differs to sigmoid neurons where small change results in only a small change in the output. it does this by allowing for the input of fractional values instead of binary inputs. The output of sigmoid neurons for i inputs is given by equation 2.

$$\Delta W = \frac{1}{1 + exp(-\Sigma_j w_j x_j - b)} \tag{2}$$

Using this equation, an output from a neuron can be non binary, smoothed out form of the step function used in perceptron neurons and will depend on an exponential function whose output will be determined by the input parameters.

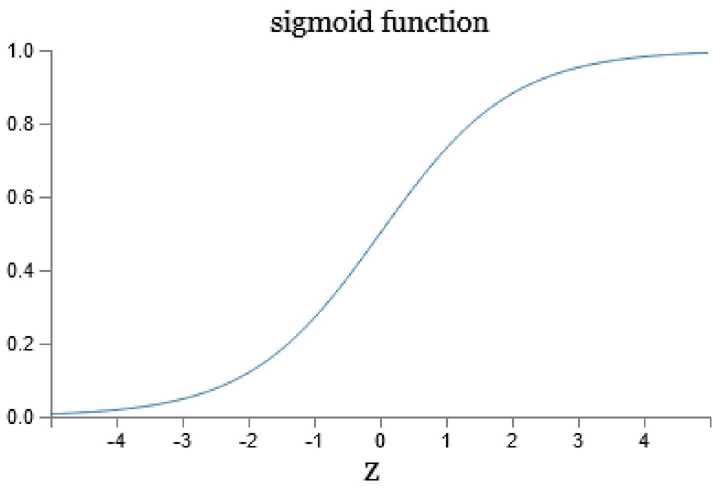

Figure 5: Sigmoid function plot [Nielsen, 2015]

3.4 Feedforward vs backpropagation

Neural networks can differ depending on if they use feedforward or backpropogation methods. Feedforward describes the process by which one neuron will relate to the others. In this type of network the flow of information is always to the right so that loops cannot form in the data and cannot connect to nodes where data was already processed through. Figure 6 shows an example of this where no arrows can pass backwards.

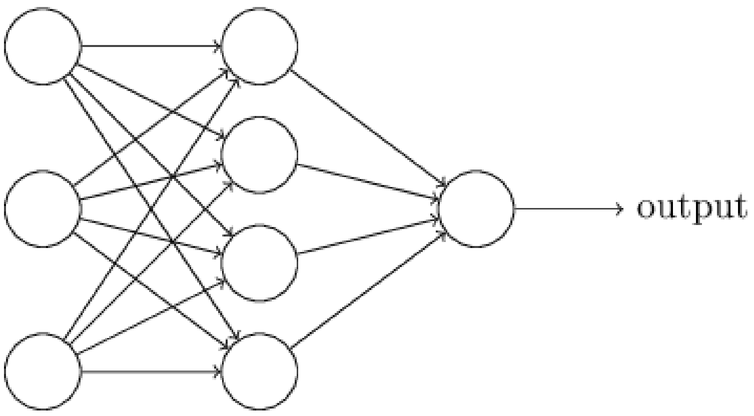

Figure 6: Feedforward neural network example [Nielsen, 2015]

Backpropogation in a neural network is where information can pass backwards. Error information from neurons further in the process is sent backwards which allows for efficient computation of the gradient at each layer. Figure 7 shows this backwards flow of information with the red arrows.

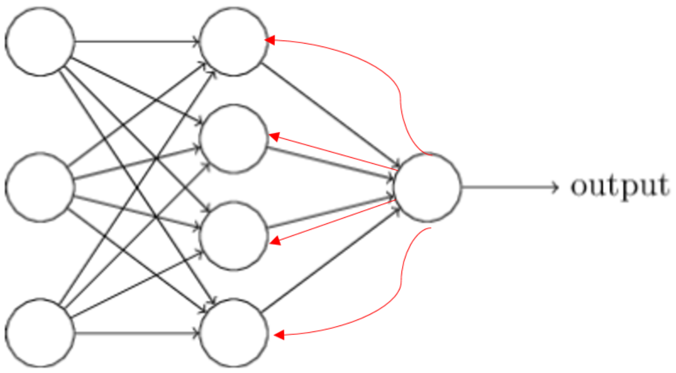

Figure 7: Backpropogation neural network shown by red lines [Nielsen, 2015].

In this network a Feedforward process is being used [Nielsen, 2015].

4 Results

4.1 Success parameters

In order to evaluate the hyperparameter variation it is important to define how successful the output is. These results come from training the neural network using 40000 data samples. This training results in a weighting associated with each neuron. Once the weights have been assigned, they are tested using testing data including 10000 data samples. The accuracy of the sample is determined as the number of successfully identified symbols opposed to the number of falsely identified symbols. The ideal accuracy is 100% for a sample. Another success parameter is the cost which is calculated by equation 3. The cost is a measure of how well the weights have been assigned to each neuron, with an ideal cost being 0 [Nielsen, 2015].

$$C(w,b) = \frac{1}{2n}\Sigma_x||y(x) - a||^2 \tag{3}$$

4.2 Default run

Initially, the neural network was run with default values. These values were set to approximate values for each hyperparameter as defined in Nielsen [2015]. These default values are, mini batch size is 10, learning parameter as 3, number of epochs is 10 and the number of hidden layers is 20. In the further analysis one of the hyperparameters is varied while the others are kept at the default values.

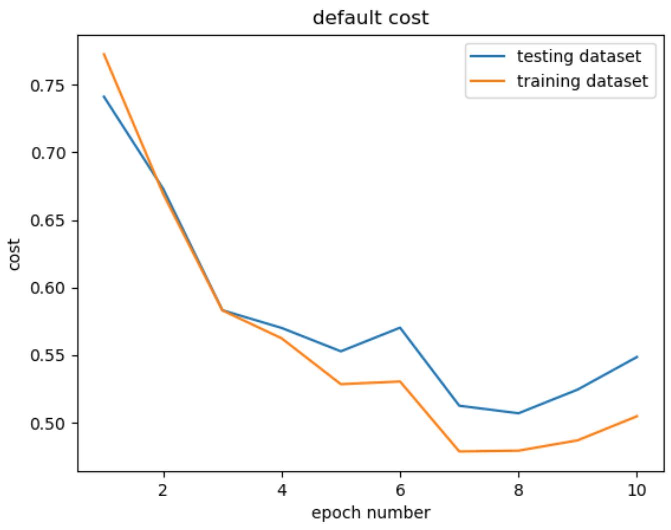

Figure 8: Graph of cost for each epoch using default parameters

Figure 8 shows how the cost varies as the number of epochs increases. As expected the cost decreases as the number of epochs increases, this is because the neural network is approaching the ideal value of 0 for cost as the number of passes through the network increases. The cost value appears to reach a minimum value of approximately 0.45 after 7 epochs.

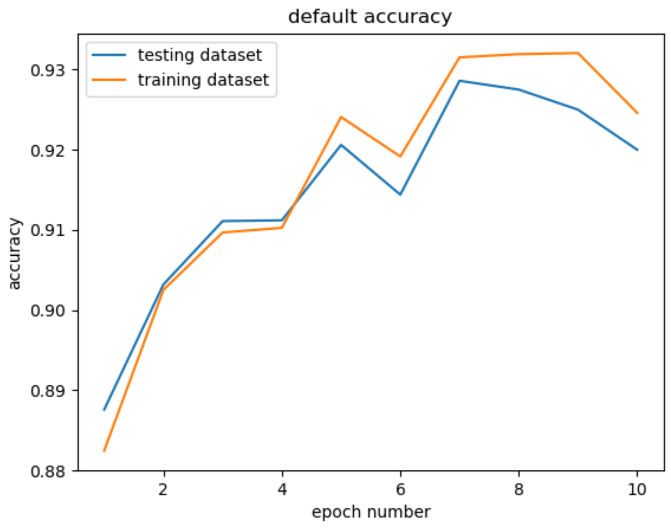

Figure 9: Graph of accuracy for each epoch using default parameters

Similar to figure 8, figure 8 shows the variation with increased epoch number. The accuracy value increases in this case, which is expected as each pass increases how well the neural network works. The accuracy appears to reach a maximum value of approximately 0.93 after 7 epochs.

4.3 Variation of learning parameter (η)

The first parameter varied is the learning parameter. figure 10 shows how the accuracy and cost vary with epoch.

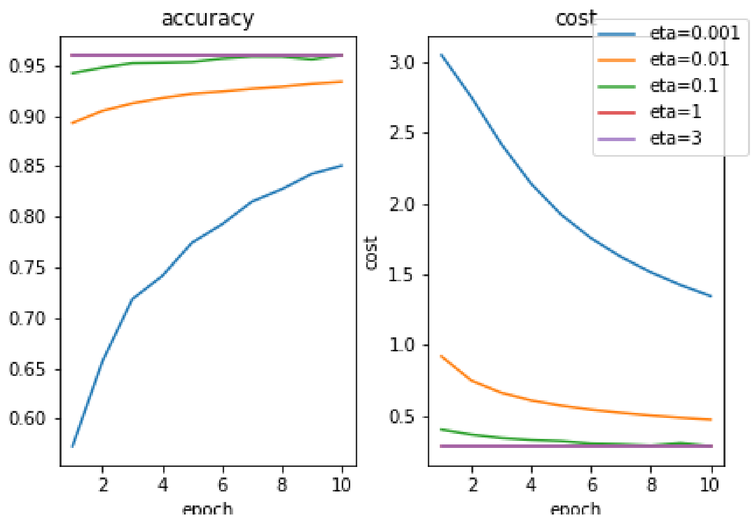

Figure 10: Graph showing how different eta values affect the cost and accuracy in each epoch

From these graphs you can see an optimum value of 0.96 for the accuracy and 0.2 for the cost is eventually reached, however the convergence rate of these differs depending on the η value. η=3 seems to reach the value the fastest so can be assumed to be an optimum value. Figure 11 shows how the accuracy value changes with an overestimate for $\eta = 10$

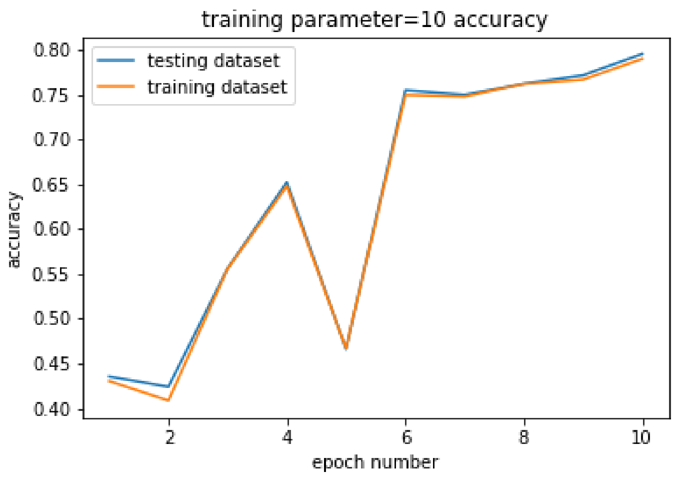

Figure 11: $\eta = 10$ accuracy with different epochs

From this graph you can see both how the accuracy value is not reached in 10 epochs, but also the large error in the 5th epoch due to overestimating the training parameter causing errors in the gradient descent by missing out maxima or minimas.
An underestimate, while stopping any of these large errors from occurring, causes a slow convergence to the optimum accuracy value, as seen in figure 12.

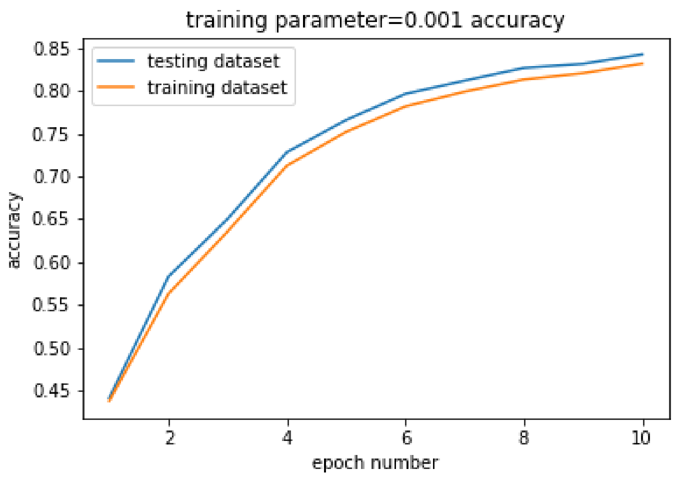

Figure 12: $\eta = 0.001$ accuracy with different epochs

4.4 Variation of mini batch size

The second parameter is the mini batch size. From figure 13 you can see the maximum accuracy value of 0.94 and cost value of 0.4.

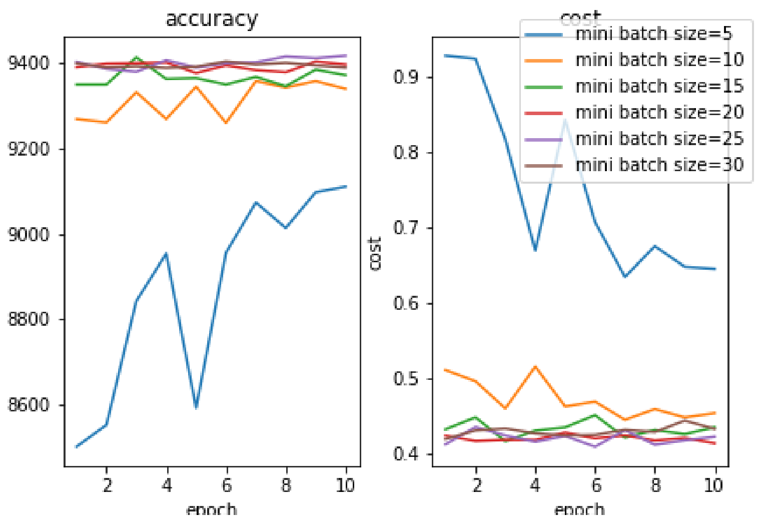

Figure 13: Graph showing how different mini batch sizes affect the cost and accuracy in each epoch, accuracy value is out of 10000

From this most of the mini batch sizes are reach quickly with a size of 20 or greater. The smaller sizes of 15 or below show how when you reduce the mini batch size too low, you sacrifice accuracy for computational efficiency. At these points you are only using data sets of size 15 or below which introduce many more statistical errors in your neural network. figure 14 Show the accuracy variation of the mini batch size of 5.

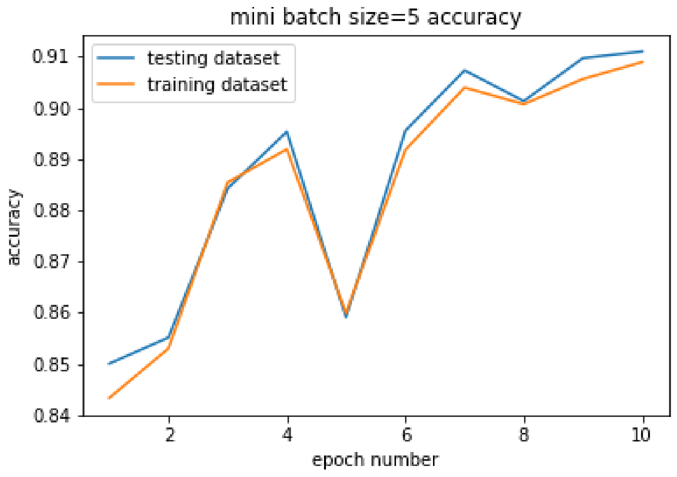

Figure 14: Mini batch size=5 accuracy variation

4.5 Variation of number of neurons

Increasing the number of hidden neurons increases the accuracy of the neural network, however this has a proportional increase on the computational time. In this analysis it is clear that figure 15 shows there is a large jump in accuracy from 5 neuron layers up to 10 neuron layers. Therefore this network requires at least 10 layers for a reasonable accuracy.

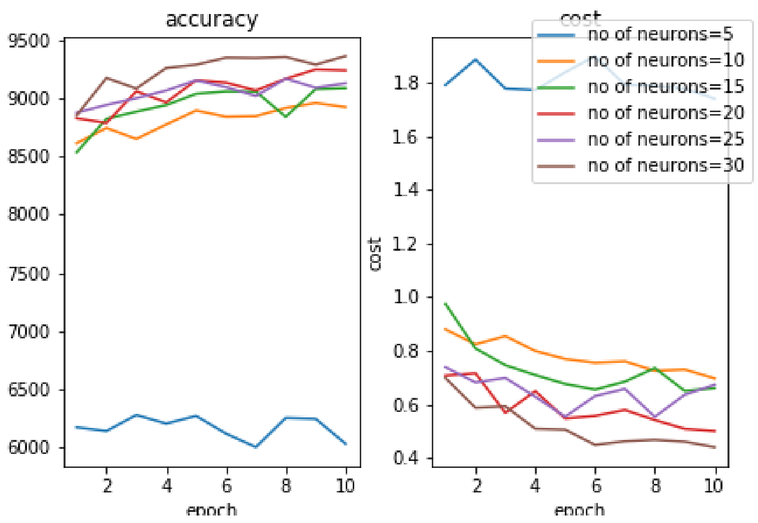

Figure 15: Graph showing how different number of neurons in the hidden layers affect the cost and accuracy in each epoch, accuracy value is out of 10000

It is also interesting to note the trend with the different number of neurons. for the 10 neurons or greater graphs there is a clear upwards trend with the increased epochs whereas with the 5 neuron system there is not. figure 16 shows this trend in more detail. In this graph the accuracy oscillates around a central value without increasing.

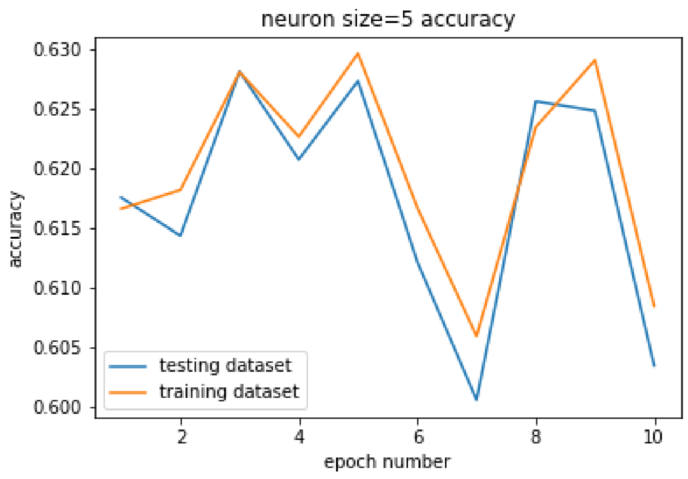

Figure 16: 5 neuron accuracy variation

5 Conclusion

In this analysis three hyperparameters for this neural network have been varied and values have been observed in each epoch. For each of these variation the accuracy and cost values have been calculated in order to evaluate how these values affect the output. A clear efficient value has been obtained for the learning parameter ($\eta = 3$) which reaches the optimum accuracy value in the least amount of epochs which can be compared to other η values which either over or underestimate this parameter. The mini batch size investigations has a clear indication of how the accuracy varies with the batch size, with the smaller size being more efficient computationally but reducing the accuracy. Clearly from this it can be seen that the mini batch size of 5 causes large error with the other values increasing in accuracy with batch size. Finally the number of neurons has small variation with accuracy when the number of hidden neurons is greater than 5, with 5 neurons causing a large drop in accuracy.
Future evaluations for this neural network could include a more quantitative approach to computational costs, directly evaluating the computation time with accuracy. From this a cost vs benefit could be looked at where some accuracy can be sacrificed for the sake of computational efficiency. Other studies could involve applying the same techniques to similar neural networks including other technical approaches, such as the use of backpropogation.

A Code

A.1 Default training

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
import matplotlib.pyplot as plt
import numpy as np

def plot_accuracy_epochs(name,ev_accuracy,train_accuracy): #plots ev and training accuracy
    against epoch no
    x=range(1,len(train_accuracy)+1)
    ev_accuracy=np.divide(ev_accuracy,10000.0)
    train_accuracy=np.divide(train_accuracy,50000.0)
    savename='plots/'+name+'_accuracy.png'
    plt.figure()
    ev,=plt.plot(x,ev_accuracy,label='ev')
    train,=plt.plot(x,train_accuracy,label='train')
    plt.xlabel('epoch number')
    plt.ylabel('accuracy')
    plt.title(name+ ' accuracy')
    plt.legend([ev,train],['testing dataset','training dataset'])
    plt.savefig(savename)

def plot_cost_epochs(name,ev_cost,training_cost): #plots ev and training cost against epoch no
    x=range(1,len(train_cost)+1)
    savename='plots/'+name+'_cost.png'
    plt.figure()
    ev,=plt.plot(x,ev_cost)
    train,=plt.plot(x,train_cost)
    plt.xlabel('epoch number')
    plt.ylabel('cost')
    plt.title(name+' cost')
    plt.legend([ev,train],['testing dataset','training dataset'])
    plt.savefig(savename)

net = network2.Network([784, 30, 10])
#default paramaters are epochs=10, mini batch size=10, learning rate=3, number of neurons=30
#each parameters is varied keeping the other parameters as defaults.
mini_batch_size_def=10
eta_def=3.0
lmbda_def=0.0
epochs_def=10

#default
ev_cost,ev_accuracy,train_cost,train_accuracy=net.SGD(training_data, epochs=epochs_def,
    mini_batch_size=mini_batch_size_def, eta=eta_def,lmbda=lmbda_def,evaluation_data=test_data
    ,monitor_evaluation_cost=True,
            monitor_evaluation_accuracy=True,
            monitor_training_cost=True,
            monitor_training_accuracy=True)
plot_accuracy_epochs('default',ev_accuracy,train_accuracy)
plot_cost_epochs('default',ev_cost,train_cost)
```

A.2 Varying training parameter

```
#variation of training parameter eta
net = network2.Network([784, 30, 10])
mini_batch_size_def=10
lmbda_def=0.0
epochs_def=10
eta=[0.01,0.1,1,10,100]
labels=[None]*len(eta)
for i in range(0,len(eta)):
    labels[i]='eta='+str(eta[i])
plt.figure()
fig, (ax1,ax2)=plt.subplots(1,2)
fig.tight_layout()
#fig.suptitle('Training paramater variation')
ax1.set_title('accuracy')
ax1.set(xlabel='epoch',ylabel='accuracy')
ax2.set_title('cost')
ax2.set(xlabel='epoch',ylabel='cost')
x=range(1,len(train_cost)+1)

for i in range(0,len(eta)):
    ev_cost,ev_accuracy,train_cost,train_accuracy=net.SGD(training_data, epochs=epochs_def,
    mini_batch_size=mini_batch_size_def, eta=eta[i],lmbda=lmbda_def,evaluation_data=test_data,
    monitor_evaluation_cost=True,
            monitor_evaluation_accuracy=True,
            monitor_training_cost=True,
            monitor_training_accuracy=True)
    ax1.plot(x,ev_accuracy,label=labels[i])
    ax2.plot(x,ev_cost)
    plot_accuracy_epochs('training parameter='+str(eta[i]),ev_accuracy,train_accuracy)
    plot_cost_epochs('training parameter='+str(eta[i]),ev_cost,train_cost)
fig.legend()
fig.savefig('plots/eta.png')
```

A.3 Varying mini batch size

```
#variation of mini batch
net = network2.Network([784, 30, 10])
mini_batch_size=[5,10,15,20,25,30]
lmbda_def=0.0
epochs_def=10
```

```
eta_def=3.0
labels=[None]*len(mini_batch_size)
for i in range(0,len(mini_batch_size)):
    labels[i]='mini batch size='+str(mini_batch_size[i])
plt.figure()
fig, (ax1,ax2)=plt.subplots(1,2)
fig.tight_layout()
#fig.suptitle('Training paramater variation')
ax1.set_title('accuracy')
ax1.set(xlabel='epoch',ylabel='accuracy')
ax2.set_title('cost')
ax2.set(xlabel='epoch',ylabel='cost')
x=range(1,len(train_cost)+1)
for i in range(0,len(mini_batch_size)):
    ev_cost,ev_accuracy,train_cost,train_accuracy=net.SGD(training_data, epochs=epochs_def,
    mini_batch_size=mini_batch_size[i], eta=eta_def,lmbda=lmbda_def,evaluation_data=test_data,
    monitor_evaluation_cost=True,
            monitor_evaluation_accuracy=True,
            monitor_training_cost=True,
            monitor_training_accuracy=True)
    ax1.plot(x,ev_accuracy,label=labels[i])
    ax2.plot(x,ev_cost)
    plot_accuracy_epochs('mini batch size='+str(mini_batch_size[i]),ev_accuracy,train_accuracy
    )
    plot_cost_epochs('mini batch size='+str(mini_batch_size[i]),ev_cost,train_cost)
fig.legend()
fig.savefig('plots/mini_batch.png')
```

A.4 Varying number of neurons

```
#variation of number of neurons
neuron_size=[5,10,15,20,25,30]
mini_batch_size_def=10
lmbda_def=0.0
epochs_def=10
eta_def=3.0
labels=[None]*len(neuron_size)
for i in range(0,len(neuron_size)):
    labels[i]='no of neurons='+str(neuron_size[i])
plt.figure()
fig, (ax1,ax2)=plt.subplots(1,2)
fig.tight_layout()
#fig.suptitle('Training paramater variation')
ax1.set_title('accuracy')
ax1.set(xlabel='epoch',ylabel='accuracy')
ax2.set_title('cost')
ax2.set(xlabel='epoch',ylabel='cost')
x=range(1,len(train_cost)+1)
for i in range(0,len(neuron_size)):

    net = network2.Network([784, neuron_size[i], 10])
    ev_cost,ev_accuracy,train_cost,train_accuracy=net.SGD(training_data, epochs=10,
    mini_batch_size=mini_batch_size_def, eta=eta_def,lmbda=lmbda_def,evaluation_data=test_data
    ,monitor_evaluation_cost=True,
            monitor_evaluation_accuracy=True,
            monitor_training_cost=True,
            monitor_training_accuracy=True)
    ax1.plot(x,ev_accuracy,label=labels[i])
    ax2.plot(x,ev_cost)
    plot_accuracy_epochs('neuron size='+str(neuron_size[i]),ev_accuracy,train_accuracy)
    plot_cost_epochs('neuron size='+str(neuron_size[i]),ev_cost,train_cost)
fig.legend()
fig.savefig('plots/neuron_size.png')
```

References

Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA:, 2015.

Shuai Zheng, James J Lu, Nima Ghasemzadeh, Salim S Hayek, Arshed A Quyyumi, and Fusheng Wang. Effective information extraction framework for heterogeneous clinical reports using online machine learning and controlled vocabularies. *JMIR medical informatics*, 5(2):e12, 2017.

scikit. Batch gradient descent vs stochastic gradient descent. `https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php`. Accessed: 10/29/2019.

Sebastian Raschka. *Python machine learning*. Packt Publishing Ltd, 2015.