2802ICT Assignment 2

# Introduction

The purpose of this document is to provide my understanding of the mechanisms of a neural network, including forward and back propagation, and to compare the results of a digit classification by implementing this with different hyper-parameter settings. To enhance the performance, Kaiming initialisation (He et al., 2015) and learning rate decay are also implemented in this assignment.

The assignment is structured as follows: First, manual calculation of the simple neural network model is undertaken. Second, the network is implemented in Python to perform a digit recognition task based on mnist data. The implementation method is addressed while the results with the default hyper-parameters and with different settings are compared in this section. Lastly, the results with the cross-entropy loss function is compared to those with the quadratic cost function.

# Part 1: Manual calculation for a small neural network

The forward and backward processes in a small neural network is manually calculated using a calculator. The forward process consists of the linear and sigmoid functions for a hidden layer and an output layer.

The first forward process is a linear function as follows:

The manual calculation of the sample X1 and X2 is:

Then, results are passed through a sigmoid function:

It is noted that the results from the hidden layer of the second input of and the first input of is same till .

With the similar process, the results of the output layer are as follow:

After the forward process, the final loss based on the quadratic cost function is calculated.

)

With the loss of , the backpropagation results of the output layer are calculated as follow:

Individually, the chain rule above can be broken down to the following three forms:

Calculating all this together provides:

Calculating all this together provides:

After one manual backpropagation, the gradients are adjusted as follows:

# part 2: Implementation in Python

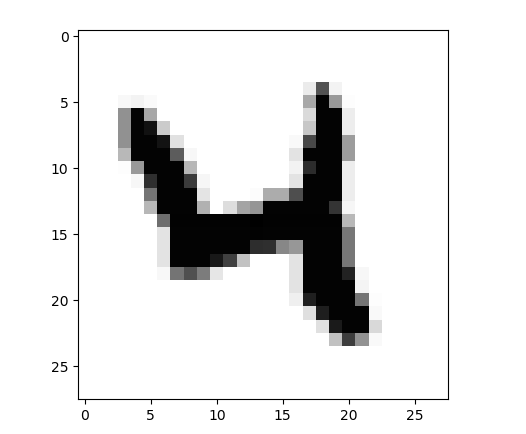
This part addresses the methods and results of the implementation in Python with the quadratic cost function.

## Implementation

The implementation of the neural network is done through matrix calculation for each batch. In some cases, an element-wise multiplication is used while a dot product is also utilised during the backpropagation process where two examples in a batch need to be added together.

Input matrix’s first dimension is the example size in a batch (e.g. 784 in mnist or 2 in a test case). Output matrix’s second dimension is the number of labels (e.g. 10 in mnist or 2 in a test case). The example of the input can be visualised by using the “visualise\_individual\_data” function, displaying the digit in a plot (see Figure 1).

Figure 1. An example of the input data in a plot



Input values are shuffled so that different digit numbers can be mixed within a batch. Without standardising input values, the learning process is too slow. Thus, input values are standardised around mean 0.0 and standard deviation 1.0 for each batch prior to forwarding the network.

Similarly, without standardising weights and biases, the learning process is tedious. Therefore, random weights and biases are initialised with mean 0.0 and standard deviation 1.0. To aim for a better initialisation, Kaiming initialisation is implemented by multiplying to the random initialisation. The comparison of the performance is listed in the result section.

A function to reduce the learning rates as time passes is implemented to ensure that the gradient steps are smaller toward the optimum point. Output labels are one-hot encoded to compare with the prediction.

## Comparison with manual calculation

Testing to see whether the network is implemented correctly is undertaken by comparing the results of the manual calculation to the outputs of the network implemented in Python below. Except for the different precision, the results are consistent between the manual calculation and the outputs from the implementation.

| Result | Manual calculation | Output |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Results

### result with the default hyper-parameters

The default hyper-parameters for the result includes:

the number of input = 784

the number nodes in a hidden layer = 30

the number of output = 10

the number of epochs = 30

the number in a batch = 20

learning rate = 3.0

cost function: the quadratic cost function

Table 1. Accuracy result with the aforementioned default hyper-parameters

|  |  |  |
| --- | --- | --- |
| Result | Random initialisation | Kaiming initialisation |
| Maximum accuracy | 94.4400 % | 94.8100% |

Given all other parameter are equivalent, Kaiming initialisation provides marginally better accuracy. The network with Kaiming initialisation starts at higher accuracy and improves more quickly. However, regardless of different initialisation methods, the gap between the training and test accuracy becomes wider as the number of epochs increases. This may be due to overfitting starting to occur with the training data.

Figure 2. Accuracy over epoch – random initialisation

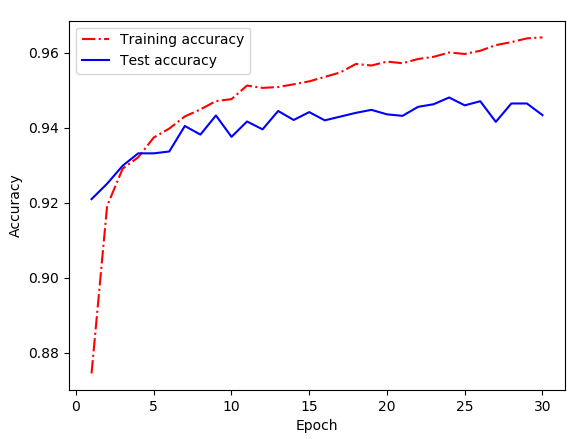
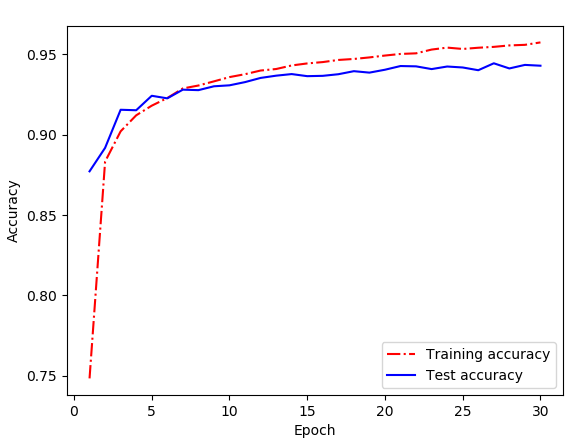


Figure 3. Accuracy over epoch – Kaiming initialisation



### Different learning rates

This section documents the experiments with different learning rates. The networks with random and Kaiming initialisation are examined.

With a small learning rate (lr) such as 0.001, the learning process is very slow (in particular, for the model with random initialisation) and it still displays underfitting symptoms after 30 epochs for both models. The lr 0.1 improves slower than the performance with the learning rate 3.0 but this learning rate seems to be appropriate as the accuracy reaches to a higher performance compared to the performance of 95.4900% with lr = 3.0 for the network with Kaiming initialisation.

However, when lr is 100, the network performance does not improve better than that of the smaller learning rates for both models. Lr of 100 performs terrible because the network cannot perform the gradient descent properly. With the too large learning rate, the model’s convergence cannot be made as it only goes back and forth near the similar points instead of descending towards the optimum.

Figure 4. Conceptual representation of the movement of the large learning rate

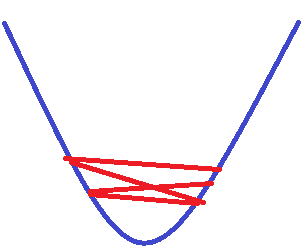
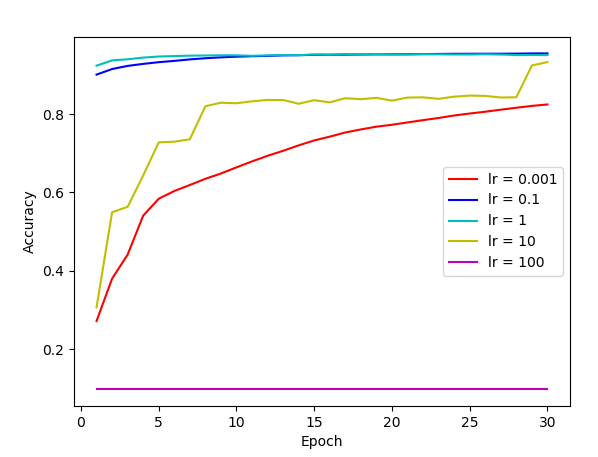


Table 2. Accuracy result with different learning rates

| Learning Rate | Random initialisation | Kaiming initialisation |
| --- | --- | --- |
| 0.001 | 21.8700% | 82.4600% |
| 0.1 | 88.3700% | 95.4900% |
| 1.0 | 93.0900% | 95.3300% |
| 10 | 94.1700% | 93.2800% |
| 100 | 10.9600% | 9.7600% |

Overall, the learning rate of 0.1 and 1 provides the better accuracy for the model with Kaiming initialisation while the learning rate of 1 and 10 works better for the model with random initialisation.

Figure 5. Accuracy over epoch with different learning rates – Kaiming initialisation



### mini-batch sizes

This section addresses the experiments with different batch sizes. The networks with random and Kaiming initialisation are compared.

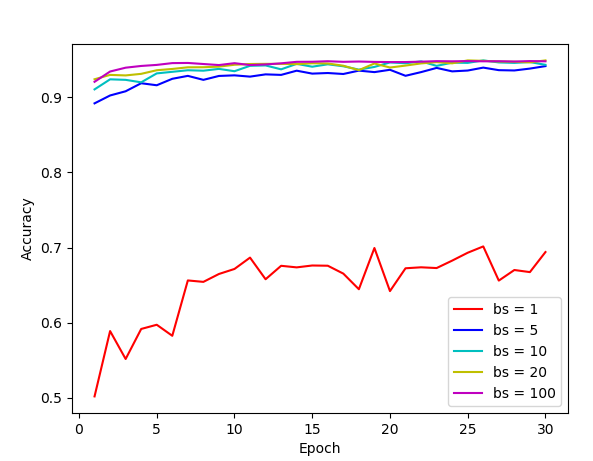
For both networks, the batch size of 1 displays a slow learning process and underfitting behaviours. For both models, as the size of each batch grows, the accuracy improves marginally until the batch size of 20. The best accuracy is shown with the batch size of 10 or 20. However, the batch size of 100 displays a poorer performance for both models.

Table 3. Accuracy result with different batch sizes

|  |  |  |
| --- | --- | --- |
| Batch size | Random initialisation | Kaiming initialisation |
| 1 | 62.5500% | 70.1500% |
| 5 | 93.4200% | 94.1600% |
| 10 | 94.5700% | 94.9200% |
| 20 | 93.9700% | 94.9200% |
| 100 | 91.8700% | 94.8300% |

Overall, the model with Kaiming initialisation seems to deal with the larger the batch size better.

Figure 6. Accuracy over epoch with different batch sizes – Kaiming initialisation



### results with combinations of different hyper-parameters

This section provides the results of the combinations of different hyper-parameters for the model with Kaiming initialisation. The quadratic cost function is used for all cases.

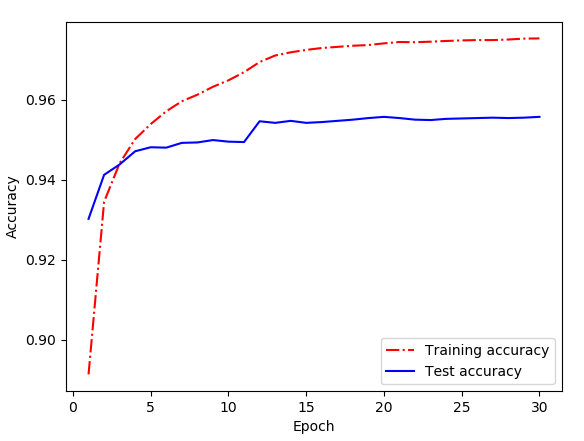
To enhance the maximum accuracy, the learning rate decay mechanism is adopted. The learning rate is decreased every 10th epoch to ensure when it approaches the optimal point, the gradient descent takes a smaller step so that it will not pass the optimum.

Table 4. Maximum accuracy with different combinations of hyper-parameters

|  |  |
| --- | --- |
| Combination | Kaiming initialisation |
| Default: epochs=30, lr=3.0, bs=20, no lr decay | 94.8100% |
| epochs=30, lr=3.0, bs=20, lr decay | 95.0400% |
| epochs=30, lr=1.0, bs=20, lr decay | 95.5600% |
| epochs=30, lr=0.5, bs=20, lr decay | 95.3700% |
| epochs=30, lr=1.0, bs=25, lr decay | 95.5700% |
| epochs=30, lr=1.0, bs=50, lr decay | 95.4100% |
| epochs=30, lr=1.0, bs=100, lr decay | 95.1300% |

In summary, the maximum accuracy, 95.5700% can be achieved with the settings of epochs=30, lr=1.0, bs=25, and lr decay. However, the overfitting pattern seems to be visible around this maximum accuracy.

Figure 7. Accuracy over epoch with epochs=30, lr=1.0, bs=25, and lr decay.



# Part 3: An alternate cost function

### results with the default hyper-parameters

The default hyper-parameters for this result equal to those from the model with the quadratic cost function. However, in this experiment, the cross-entropy cost function is used.

The both networks achieve a similar accuracy with the cross-entropy cost function. Despite only a minor improvement in the accuracy for the model with Kaiming initialisation, according to Nielsen (2018)[[1]](#footnote-1), the cross-entropy cost function can be resilient to the issues associated with the quadratic cost function’s learning slowdown when using many-layer and multi-neuron network. Hence, using this cost function makes sense when there is a neural network with many layers and nodes.

Table 5. Maximum accuracy with the aforementioned default hyper-parameters

|  |  |  |
| --- | --- | --- |
| Maximum accuracy | Random initialisation | Kaiming initialisation |
| Quadratic cost function | 94.4400 % | 94.8100% |
| Cross-entropy cost function | 94.1300% | 94.9000% |

Figure 8. Accuracy over epoch – random initialisation, cross-entropy cost function

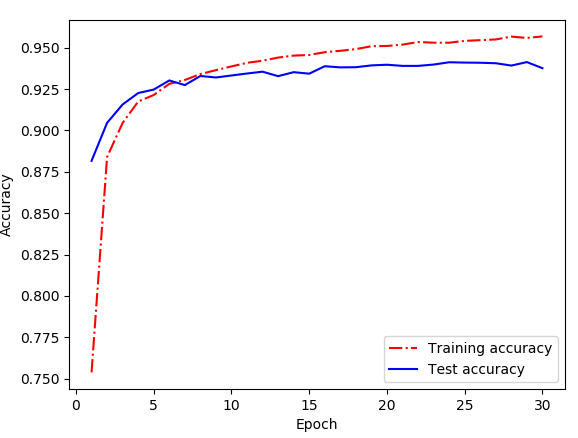
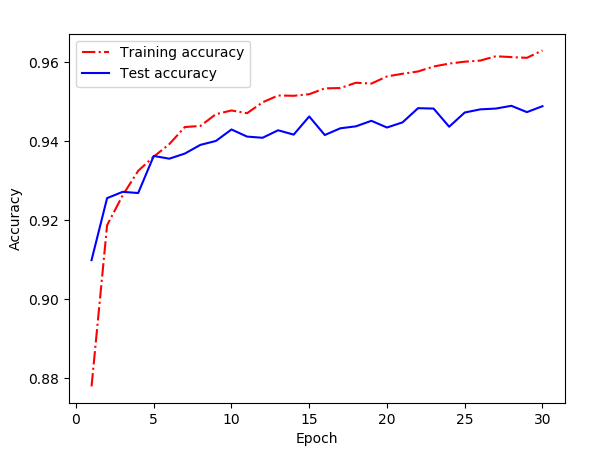


Figure 9. Accuracy over epoch – Kaiming initialisation, cross-entropy cost function



### results with combinations of different hyper-parameters

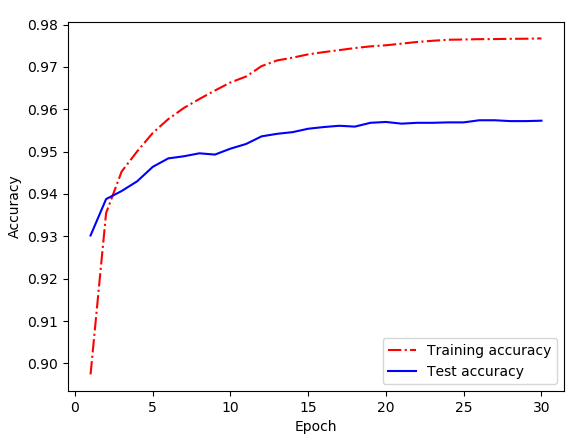
This section shows attempts to enhance the maximum accuracy with additional changes to the hyper-parameters: the cost function and the number of epochs. The best accuracy is achieved through the increased hidden nodes of 90 instead of 30.

Table 6. Maximum accuracy with different combinations of hyper-parameters

|  |  |
| --- | --- |
| Combination | Kaiming initialisation |
| Best: epochs=30, lr=1.0, bs=25, lr decay with the quadratic cost function | 95.57% |
| epochs=30, lr=5.0, bs=20, lr decay with the cross-entropy cost function | 94.74% |
| epochs=30, lr=3.0, bs=20, lr decay with the cross-entropy cost function | 95.26% |
| epochs=30, lr=1.0, bs=20, lr decay with the cross-entropy cost function | **95.74%** |
| epochs=30, lr=0.5, bs=20, lr decay with the cross-entropy cost function | 95.71% |
| epochs=30, lr=1.0, bs=25, lr decay with the cross-entropy cost function | 95.46% |
| epochs=30, lr=1.0, bs=20, lr decay, hidden nodes=60 with the cross-entropy cost function | 96.87% |
| epochs=60, lr=1.0, bs=20, lr decay, hidden nodes=60, with the cross-entropy cost function | 96.89% |
| epochs=60, lr=1.0, bs=20, lr decay, hidden nodes=90, with the cross-entropy cost function | **97.20%** |

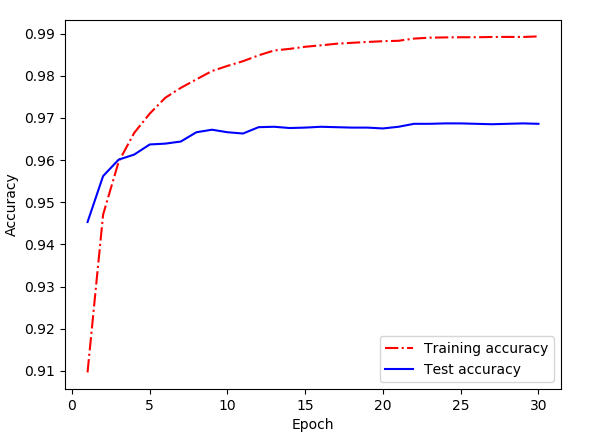
Without increasing the number of nodes in a hidden layer, the maximum accuracy is 95.74% with epochs=30, lr=1.0, bs=20, and lr decay with the cross-entropy cost function, as shown below.

Figure 10. Accuracy over epoch with epochs=30, lr=1.0, bs=20, and lr decay with the cross-entropy cost function



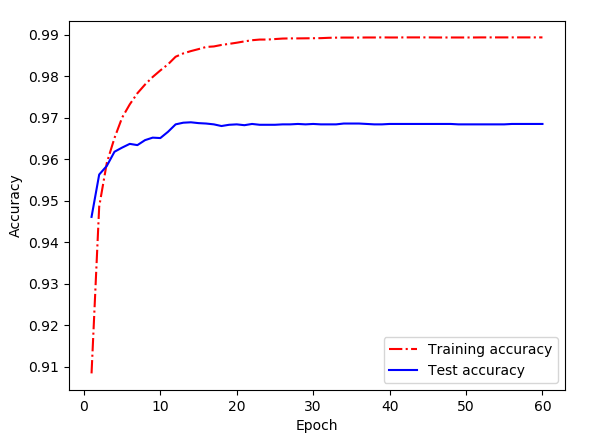
The larger the number of hidden nodes, the better the maximum accuracy is.

Figure 11. Accuracy over epoch with epochs=30, lr=1.0, bs=20, lr decay, hidden nodes=60, with the cross-entropy cost function



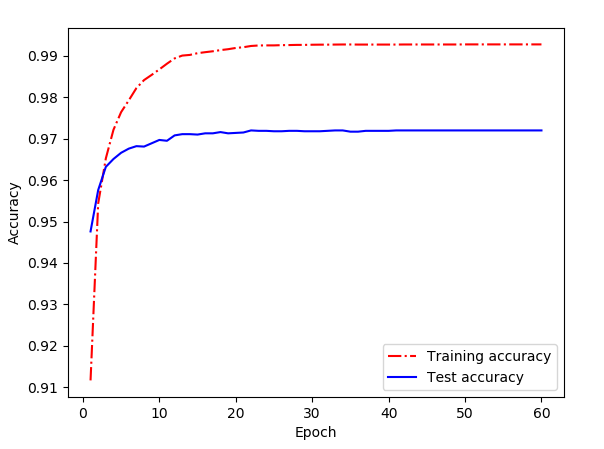
Increasing the number of epochs does not seem to make a larger difference due to overfitting.

Figure 11. Accuracy over epoch with epochs=60, lr=1.0, bs=20, lr decay, hidden nodes=60, with the cross-entropy cost function



However, the increased hidden node of 90 further provides better accuracy of 97.20% on the test dataset. The accuracy of 97.20% is reached within 30 epochs, thus, the number of epochs over 30 does not make a difference.

Figure 12. Accuracy over epoch with epochs=60, lr=1.0, bs=20, lr decay, hidden nodes=90, with the cross-entropy cost function



# Conclusion

In summary,

1. http://neuralnetworksanddeeplearning.com/chap3.html [↑](#footnote-ref-1)