TBMI26
Assignment 1

Martin Estgren <mares480>

February 12, 2017

1 Assignment 1

1.1 Overview of the data

Looking at the data from a machine learning perspective we can observe how the data is compressed from $32 * 32$ bitmap images to 64 features with the integer range 0..16. This means that we can create a hypercube feature space with 64 dimensions. We have not lost any information but the number of dimensions have significantly decreased ($32*32*1 = 1024$ and $64*16 = 1024$).

The pre-processing of the data doesn't reduce the information in the data set but make it less noisy by introducing some feature-invariance.

1.2 Implementation of the kNN algorithm

1.2.1 kNN without cross-validation

The implementation of kNN is fairly straight forward.

We iterate though all the cases in the test set and calculate the distance to each point in the training set. The result is then sorted in ascending order and the k first elements are counted in regards to what label they are assigned. The label with the most values are then picked to be assigned for the testing case we are currently processing. When no counted label is higher than another we pick the label with the data point which have the closest euclidean distance to the test case.

The result of our implementation of the kNN algorithm with k arbitrarily chosen as 4 can be seen below.

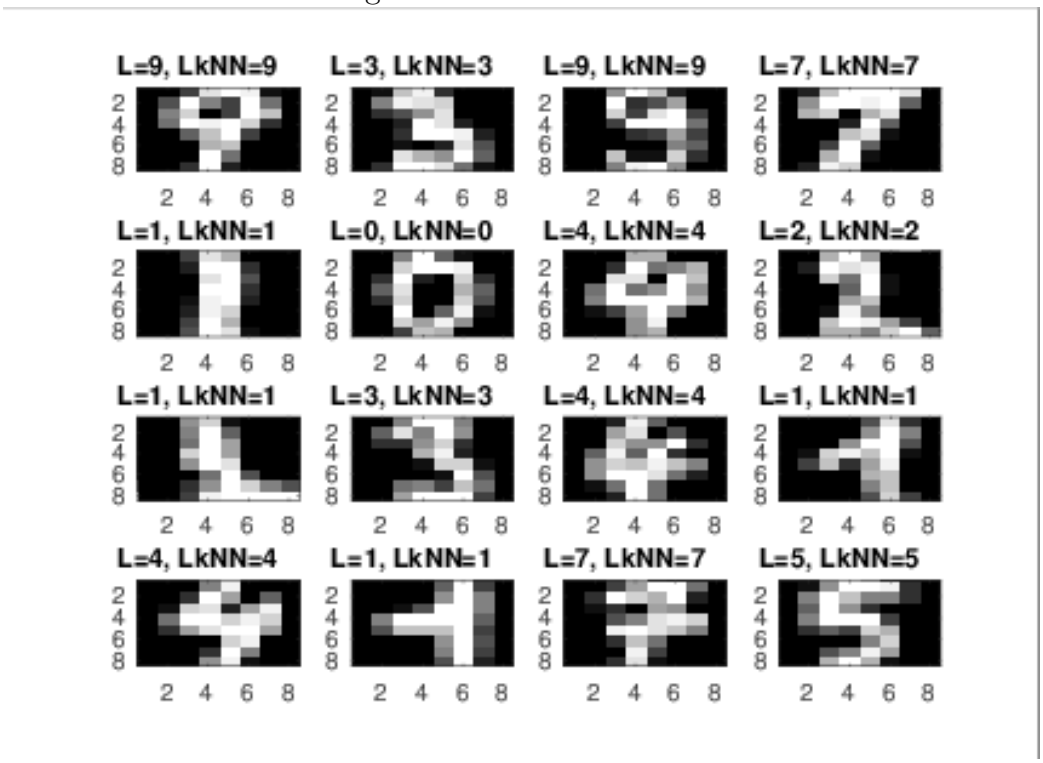
1. k = 4

2. Accuracy = 0.9710

```
cM =

    99     0     0     0     0     0     0     0     0     0
     0    97     1     0     0     0     0     0     2     3
     0     0    97     0     0     0     0     0     0     1
     0     1     0    98     0     0     0     0     0     3
     0     1     0     0    96     0     0     0     0     2
     0     0     0     1     0   100     0     0     0     1
     1     0     0     0     0     0   100     0     0     0
     0     0     2     0     0     0     0   100     0     0
     0     1     0     0     1     0     0     0    96     2
     0     0     0     1     3     0     0     0     2    88
```

Figure 1: Result from kNN



1.2.2 kNN with n-fold cross validation

For the cross validation version the n-fold cross validation algorithm was used to determine the best value for k. For all of the following results $n = 2$ was

used but the algorithm implementation allow for any value of n as long as it can evenly distribute the dataset. Accuracy is used as the validation score in order to find the best value for k.

Dataset 1

Best parameters:

- $k = 1$
- $Accuracy = 0.9900$

Figure 2: Cross validation score

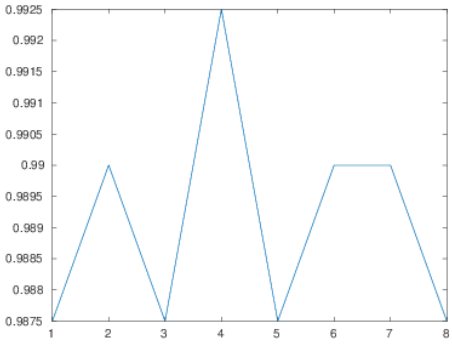


Figure 3: Cross validation result

Dataset 2

Best parameters:

- $k = 1$
- $Accuracy = 1$

Figure 4: Cross validation score

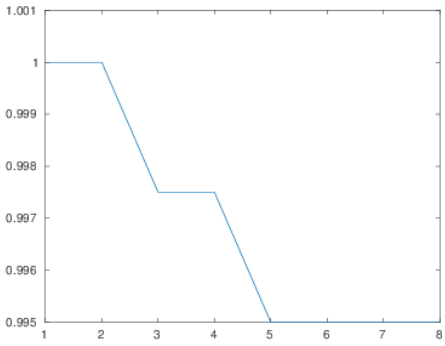
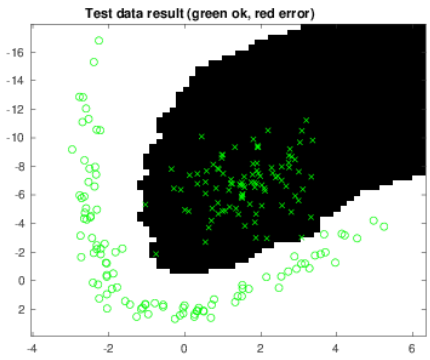


Figure 5: Cross validation result



Dataset 3

Best parameters:

- $k = 1$
- $Accuracy = 1$

Figure 6: Cross validation score

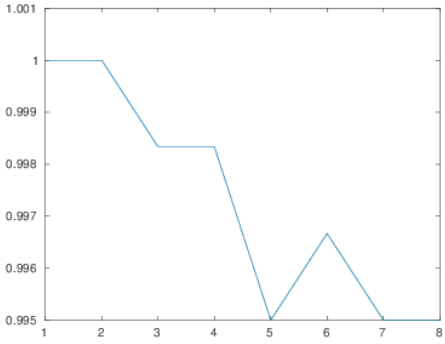
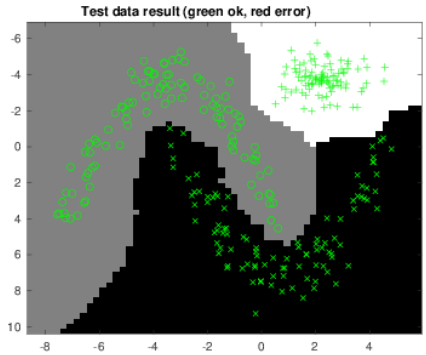


Figure 7: Cross validation result



Dataset 4

Best parameters:

- $k = 1$
- $Accuracy = 0.9840$

Figure 8: Cross validation score

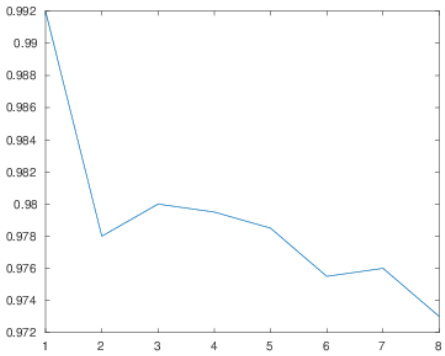
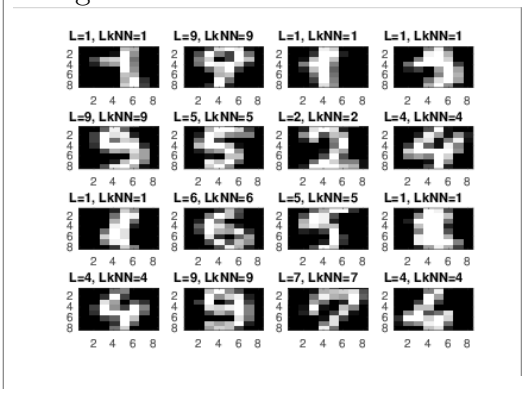


Figure 9: Cross validation result



1.3 Single/Multi-layer neural network

1.3.1 Single layer backpropagation

The implementation of the single layer backpropagation algorithm we use a linear activation function and the gradient $\frac{n}{s}(Y - Dt) * Xt^t$ where n is the number of data to classify, Y is the result from the forward propagation, Dt is the expected value for Y and Xt is a matrix with all training features. To train the network we use backpropagation with gradient descent, meaning we calculate the gradient for our output layer and use it together with our previous weights and classification error to fine-tune our weights for better classification.

Dataset 1

The classification bound of this data set is not very complex which means we don't need to find the perfect weights for the neurons. In turn it will be trained in no significant time.

Parameters:

- $Iterations = 800$
- $learningrate = 0.005$
- $Accuracy = 0.9950$

Figure 10: Single-layer network error

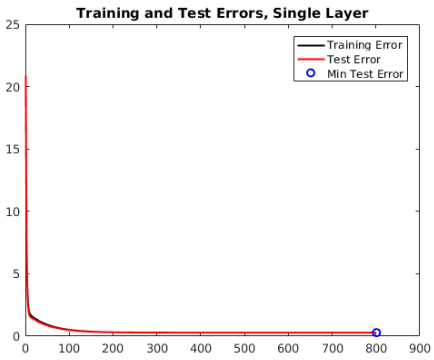
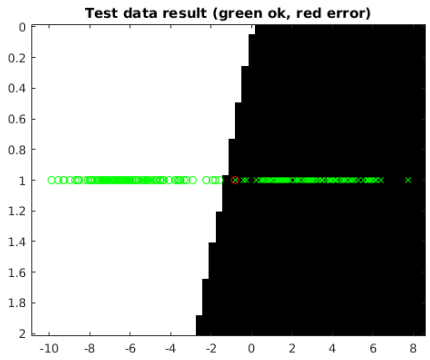


Figure 11: Single-layer network result



Dataset 2

For this dataset we require much more precision which is why we increase the number of iterations and reduced the learning rate.

Parameters:

- $Iterations = 200000$

- $learningrate = 0.00005$
- $Accuracy = 0.8200$

Figure 12: Single-layer network error

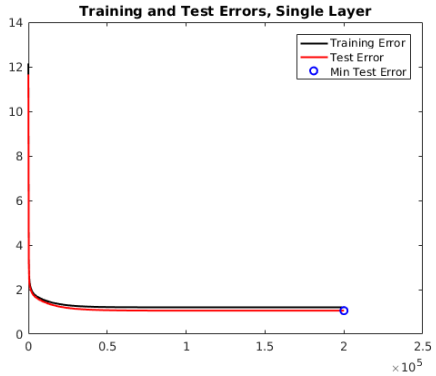
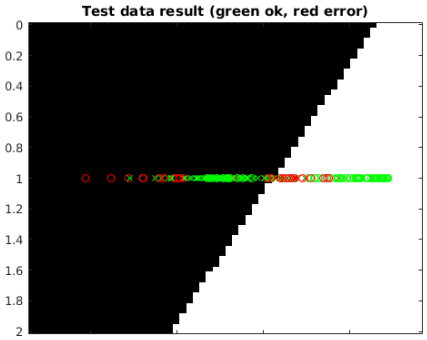


Figure 13: Single-layer network result



Dataset 3

These parameters should work just fine for this dataset as well since the data set is only slightly more complex than the last one.

Parameters:

- $Iterations = 200000$
- $learningrate = 0.00005$
- $Accuracy = 0.8533$

Figure 14: Single-layer network error

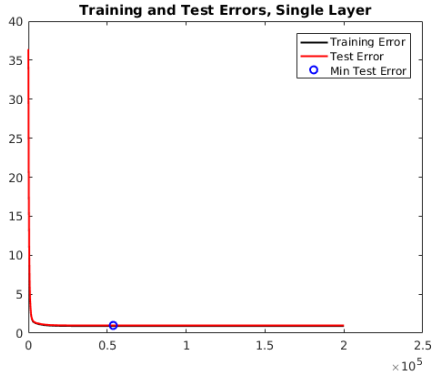
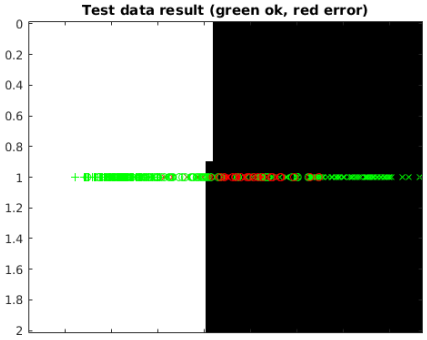


Figure 15: Single-layer network result



Dataset 4

This dataset is much more complex compared to the other three but we are still dealing with the same number of neurons. We have already seen that this learning rate and iterations result in finding the optimal weights for the parameters.

Parameters:

- $Iterations = 200000$
- $learningrate = 0.00005$
- $Accuracy = 0.9160$

Figure 16: Single-layer network error

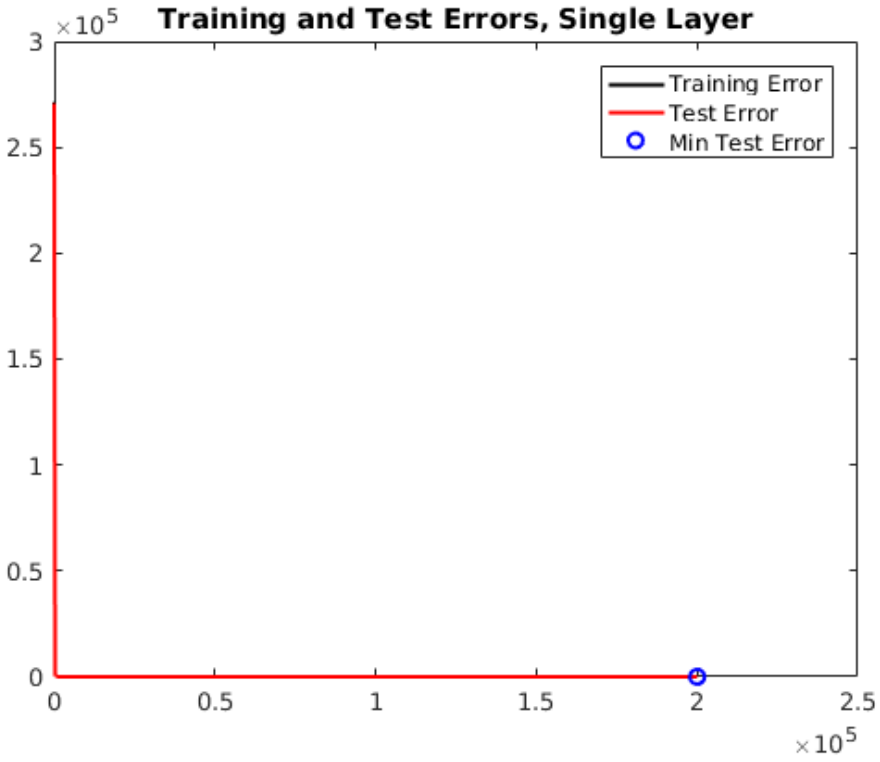
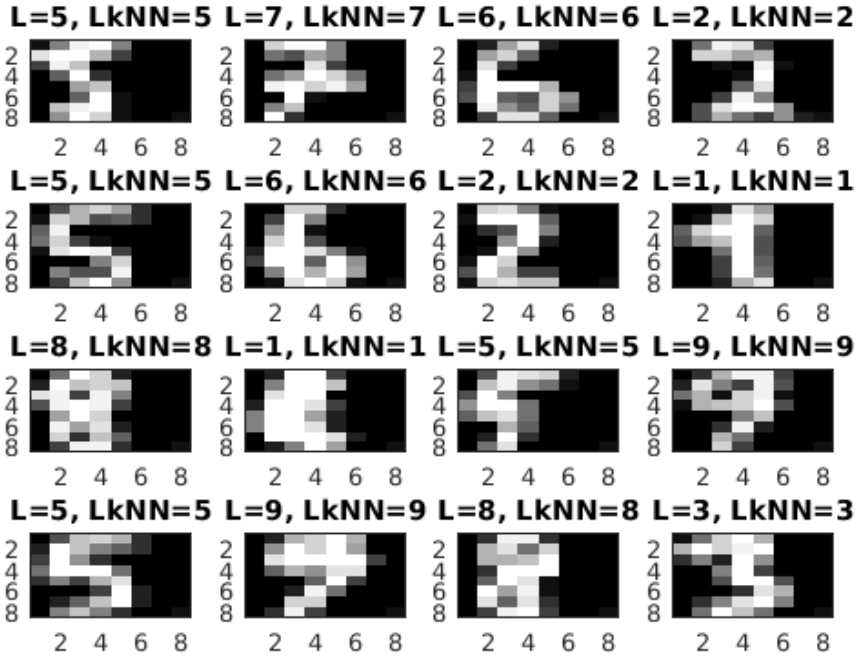


Figure 17: Single-layer network result



1.3.2 Multi-layer backpropagation

When dealing with multi layer networks we have to take care of order-of-magnitudes more number of weights. Because of this we want to limit the number of hidden neurons as much as we can. We are also using a non-linear activation function for the hidden layer which means we get transformed information at the linear output layer. As with the single-layer model we train the network using backpropagation with gradient descent. For this model we require a more elaborate gradient than the single-layer one since we have a hidden layer with non-linear activation function. To tune the hidden weights we also need to take the output layer into account, creating a chained partial derivative where both the derivative of $tanh$ and a linear function is important.

The hidden layer gradient:

$$\frac{2}{n}((V^T(Yt - Dt). * tanh'(W * Xt)) * Xt^T) \tag{1}$$

where V is the weights for the output payer and W is the weights for the hidden layer.

Dataset 1

This is a fairly simple dataset to classify with a low number of features and a linear discrimination bound already existing in feature space we only need a small number of hidden neurons.

Parameters:

- $Hidden = 5$
- $Iterations = 2000$
- $learningrate = 0.01$
- $Accuracy = 0.9950$

Figure 18: Multi-layer network error

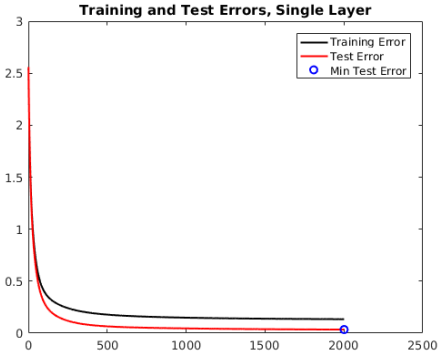


Figure 19: Multi-layer network result

Dataset 2

As with the previous dataset we have a low number of features and a fairly similar complexity for the discrimination bound. We can in this case produce a accurate representation with even fewer hidden nodes bit with a higher amount of iterations and a lower learning rate.

Parameters:

- $Hidden = 2$
- $Iterations = 30000$
- $learningrate = 0.005$
- $Accuracy = 1$

Figure 20: Multi-layer network error

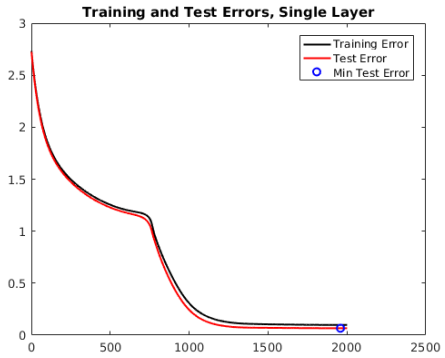


Figure 21: Multi-layer network result

Dataset 3

We are now dealing with a more complex feature space, both in terms of features and discrimination bound. Therefore we add a couple of hidden nodes and use a very small learning rate.

Parameters:

- $Hidden = 10$
- $Iterations = 4000$
- $learningrate = 0.005$
- $Accuracy = 0.9967$

Figure 22: Multi-layer network error

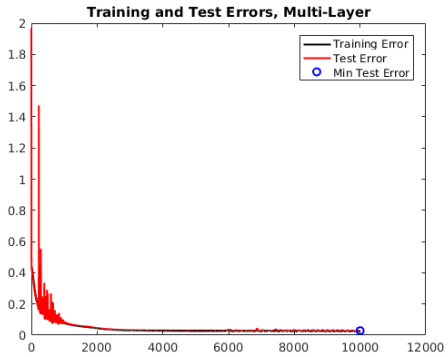


Figure 23: Multi-layer network result

Dataset 4

This dataset is a bit different, we are not dealing with only two or three dimension but 64. We also deal with more values for each features and

therefore we dramatic increase the number of hidden nodes. Thought rigours testing we find that double the number of hidden nodes in relation to the features space creates a good enough classifier.

Parameters:

- $Hidden = 128$
- $Iterations = 16400$
- $learningrate = 0.01$
- $Accuracy = 0.96498$

Figure 24: Multi-layer network error

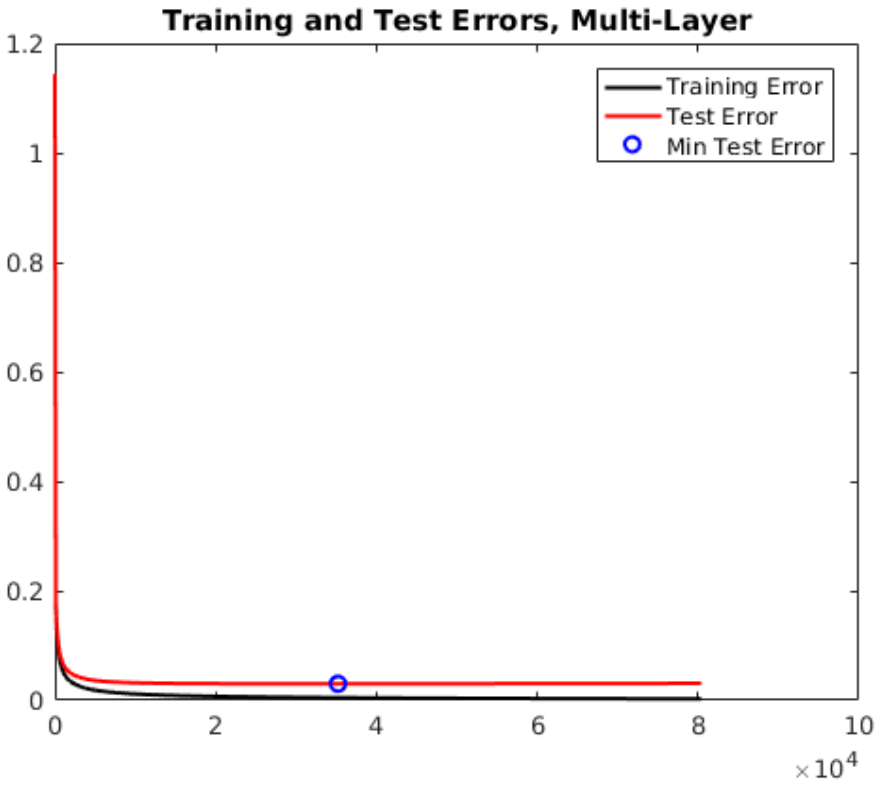
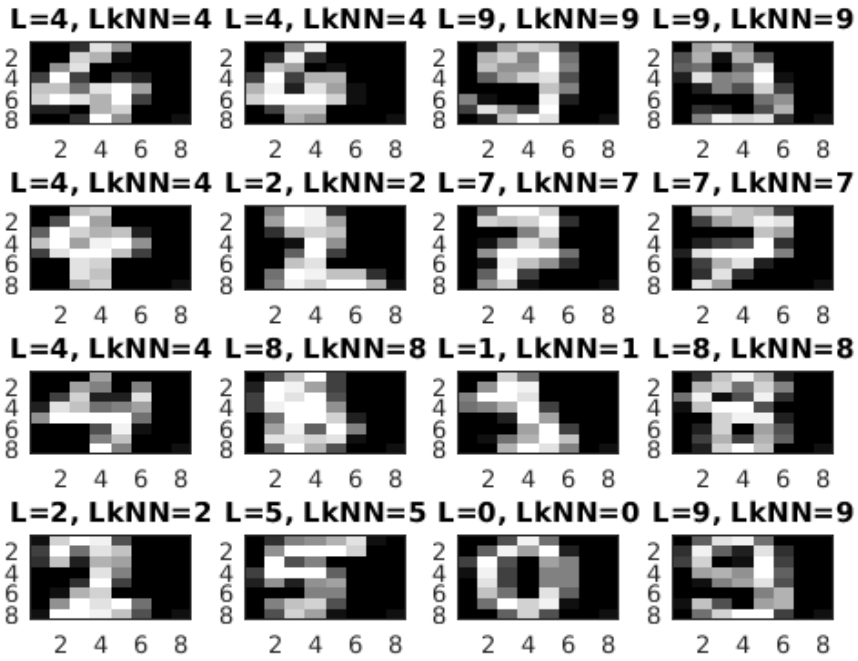


Figure 25: Multi-layer network result



1.3.3 Non-generalizable dataset

If there's enough data points in order to find a reliable model for a unknown function we say that the model we get is non-generalizable. For example we limit our testing dataset to only 500 samples out of 999 with the parameters:

- $Hidden = 10$
- $Iterations = 10000$
- $learningrate = 0.02$
- $Accuracy = 0.59059$

This training dataset is not generalizable since it doesn't contains enough data points to properly approximate a model for the true dataset.

Figure 26: Multi-layer network error

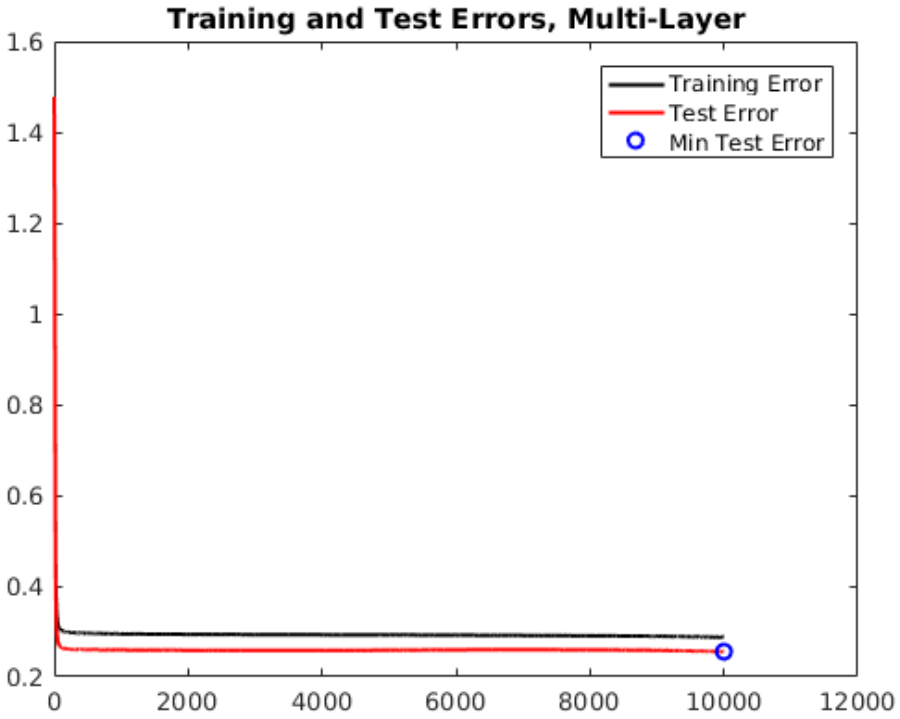


Figure 27: Multi-layer network training data

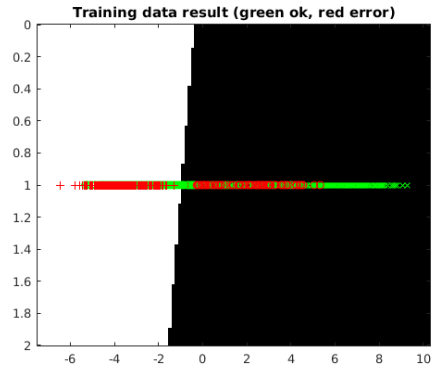
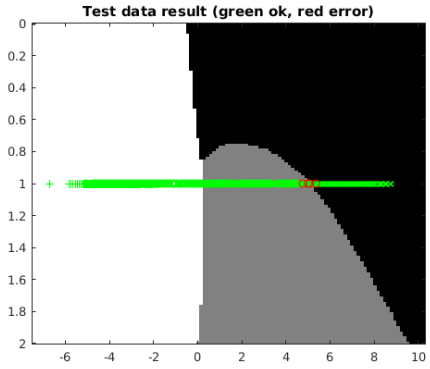


Figure 28: Multi-layer network true data



1.4 Performance of the different classifiers

kNN had a high degree of accuracy for all of the provided datasets. It's a simple algorithm which is both transparent and easy go understand. The

main drawback is that it requires that you iterate thought the entire training set for each data point we want to classify, making it bad for large datasets.

Single-layer neural network performed great on the simple datasets but had greatly reduced performance on more complex datasets. Time-wise it required order of magnitudes less training time compared to the multi-layer neural networks but at the cost of flexibility.

Multi-layer neural network had highly accurate results for each dataset but required huge amount of training time for the more complex datasets. The number of hidden nodes were fairly arbitrarily picked since we don't have the time to perform proper cross-validation. For both the single and multi-layer models, once we have a trained network - trained with a dataset similar to the true model we can easily classify new data points without consulting previous data points. This means that for huge datasets we have a significantly reduced total time in comparison with kNN.

1.5 What could improve the results

Usually when working with neural networks we try to normalize the input data before the model. This could have the potential to improve the results. We also randomly initialize the weights for the neural network models which can be replaced by ball-parking them before the training phase, in some cases significantly reducing the required model training time - which is our greatest problem with the neural networks.