Lab 1 (TBMI26) by Filip Berglund (Filbe882) and Kasper Andersson (Kasan880)

1

Data set 1: A linear classifier could work with a decent accuracy. We can almost split the data into two subsets with a straight line. However, it won't be perfect. If we want perfect separation with a straight line, another feature could be used to split the data, since they overlap now (using a straight line)

Data set 2: Here we can separate the data perfectly, however, we need a non-linear.

Data set 3: Again, we can't separate the 3 data sets perfectly, but we have to use non-linear classifiers.

2

With small images there is less room for positioning errors of the digits.

3

We start to calculating the Euclidian distance to all training samples from the data we cant to classify. We extract the indexes of the k closest training data. We check which classes the indexes belong to. Then finally we check which class gets most indexes, which results in our classification.

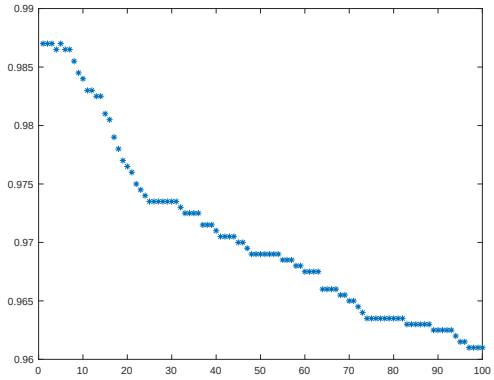
4

Since we only use k=2 classes, we can remove one data point (then one that's furthest away) to make sure we get an odd number.

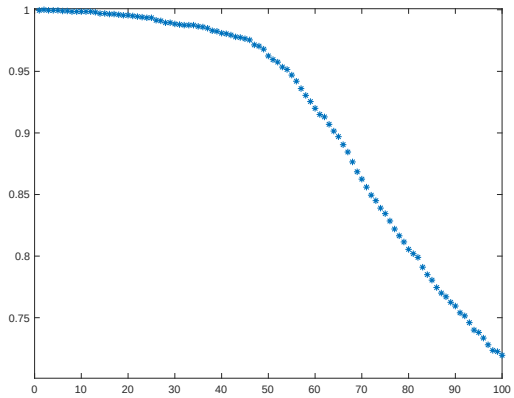
5

We looped k from 1 to 100 and we saw a pattern. The best accuracy always was when we had k between 1 and 5.

Data 1, k on x-axis, accuracy on y-axis



Data 2, k on x-axis, accuracy on y-axis



6

Single layer: We define the outputs Y for each sample as X*W, where X = the features and W = the weights. We find the max component of Y which becomes our predicted class L. To train the network we first calculate difference between predicted output Y and desired output D. Then we calculate the gradient and then we update out weights. We also added the bias as a column vector filled with ones in the input data matrix.

Multilayer: We calculate the input to the hidden layer in same way as we calculate the output in the single layer network. Then we apply the activation function tanh. Now we add a bias, same way as in single layer, then calculate the output Y as U*V, where U is the activated in data with the bias and V are the weights of the last layer. We calculate two gradients, one for each layer. For the first layer, we use

```
grad_w = -2 * XTrain'*(((DTrain - YTrain)*Vout(1:end-
1,:)').*tanhprim(XTrain*Wout)) / NTrain;
```

where XTrain is input data, `Vout(1:end-1,:)'` is the weights for the second layers without the bias, `tanhprim(XTrain*Wout)` is the activated data from the first layer to the second layer where we use tanhprim as activation function. We normalize with the number of training data, NTrain.

For the second layer, we use

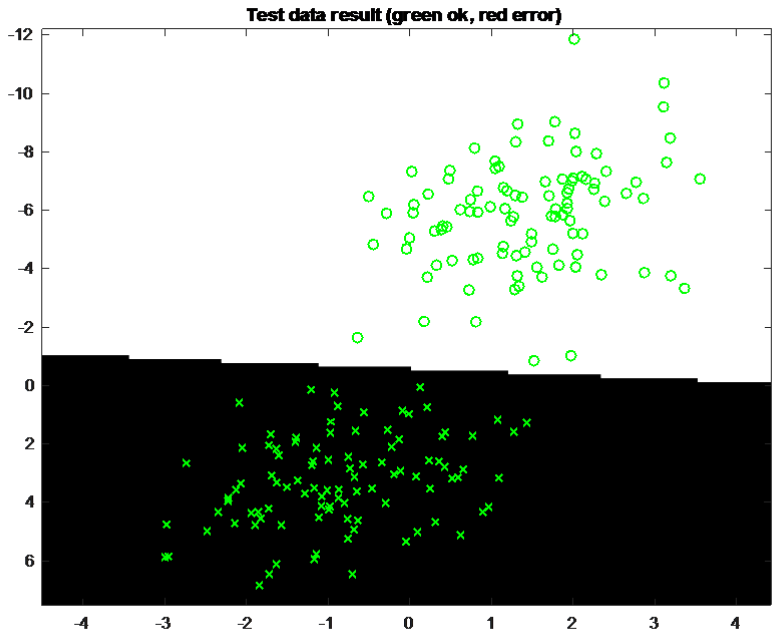
```
grad_v = -2*U'*(DTrain - YTrain) / NTrain;
```

where DTrain is the desired output and YTrain is the predicted ouput. We normalize the gradient with the number of training data, NTrain.

7

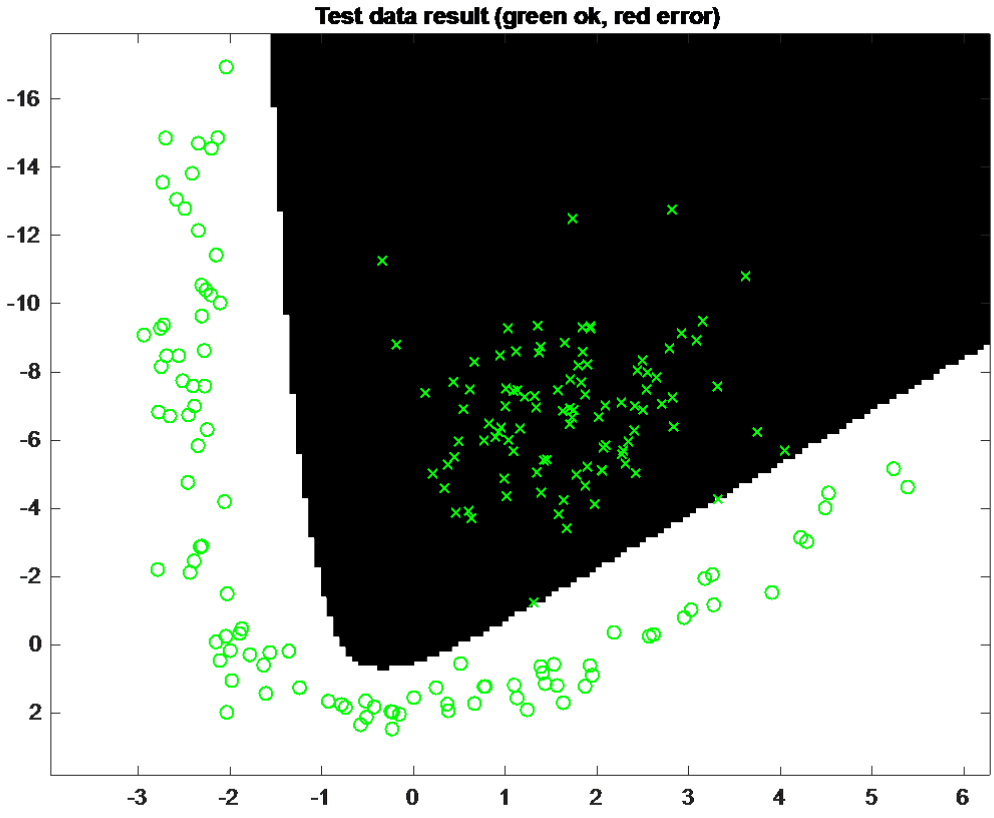
Data 1

By observing the data, we saw that it is linearly separable, so came to the conclusion that one hidden neuron is sufficient. We tested with different learning rates and number of iterations and settled on the value 200 as number of iterations and 0.01 as learning rate.



Data 2

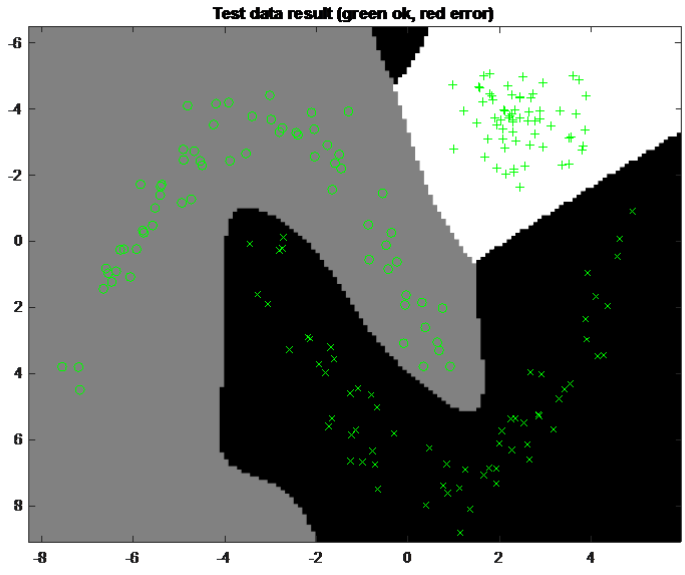
We observe that the data is not linearly separable, so we tested with two hidden neurons. We got acceptable accuracy, but the discriminant had a really bad error margin. So we tried with more hidden neurons and iterations and we got a more larger error margin, for most part of the discriminant.



50 hidden neurons, 10000 iterations and a learning rate 0.01.

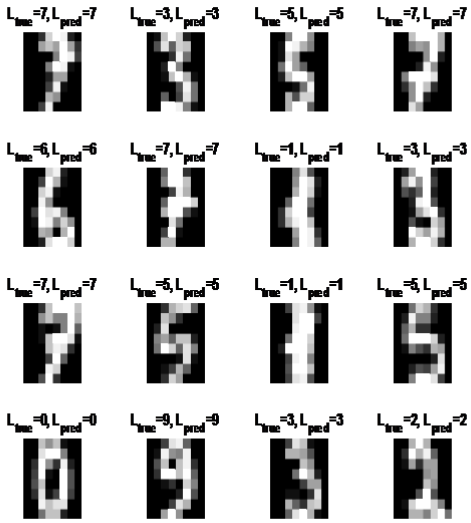
Data 3

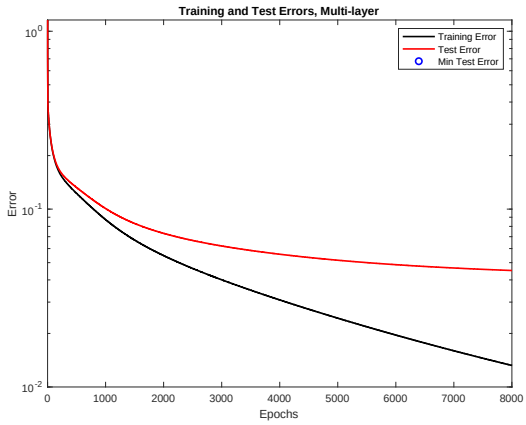
The test with difference values and we observe that with 7 hidden neurons, 8000 iterations and a learning rate of 0.02 we get 100% test accuracy many times.



Data 4

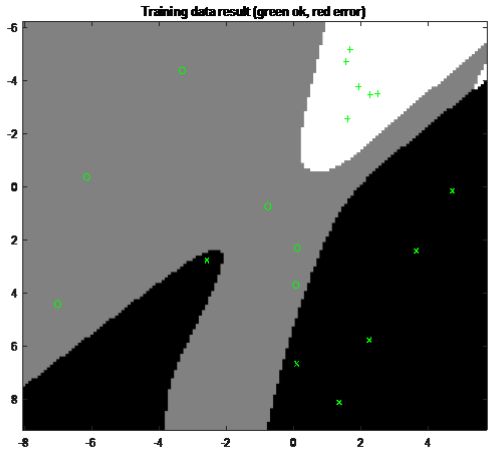
Since the dimension is much higher here, it's harder to predict how many hidden neurons we need. Also, the run time becomes way longer for this data set, which limits how many times we can run the code. We achieved a test accuracy 0.96364 with 512 hidden neurons, 800 iterations and a learning rate 0.001.



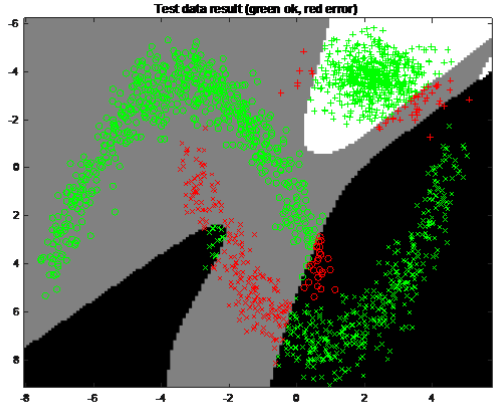


8

To make it non-generalizable, we trained the model with very few training samples but much more test data. This results in a model that fits perfectly to (in our case, about 20) the training data, but perform very badly when we run it with the test data, which is about 1900 samples.



Perfect training data result



Very bad test data result

9

kNN was super easy to implement, and it performs well and fast on smaller data sets. However, it doesn't learn after each iteration, so if the data input is too complex, we will always get bad results. Also, it performs slowly on larger high-dimensional data sets.

Single layer is simple model, with the ability the learn. Drawback is that it only can separate the data with a linear discriminant. Also, it's performs fast.

Multi-layer was the hardest to implement and requires a lot of fine tuning on the parameters. However, this results in a well-performing model that manages to separate much more complex data sets. It performs well on evaluating, but the learning process takes a lot of time when the number of hidden neurons and the number of iterations gets high.

10

For the first data set, we don't need any pre-processing, since its already linearly separable. However, the classes do overlap slightly, so perhaps adding more features would separate the classes better.

For the second data set, we need a bell shaped curve to separate the data. So a transformation on the data to make it linearly separable could aid in the learning of the network.

For the third and fourth data sets, its hard to observe anything that could be done as pre-processing. More features could help the model to separate the data points better. Perhaps higher resolution on the pictures on the fourth data set would make the predictions easier, but this would be on the cost of training time.