1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA

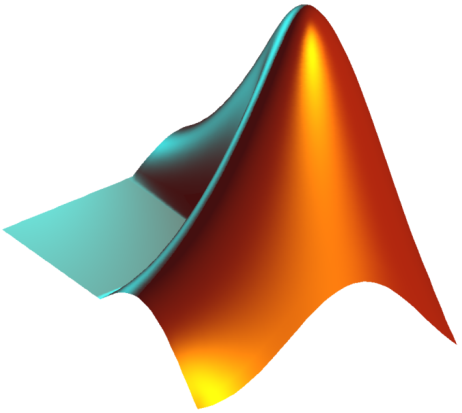
Informatics Engineering Department

Machine Learning
200/2021 - 1.º Semester

Assignment Nº1:

OCR

Optical Character Recognition



Teacher:

António Dourado Pereira Correia

Students:

Sérgio Machado, 2017265620, uc2017265620@student.uc.pt

Renato Matos, 2015257602, 2015257602@student.uc.pt

Index

Index **1**

Introduction **2**

Neural networks structure **3**

Implementation **4**

3.1 Associative memory filter 4

3.2. Perceptron as filter 4

3.3. Classifier 5

3.3. Patternnet 5

Training **6**

Results and Discussion **7**

Best Results 7

5.1. Classifier (one layer) 9

Activation function: Hardlim 9

Activation function: Purelin 9

Activation function: Logsig 10

5.2. Classifier (two layers) 10

5.3. Perceptron + Classifier 11

Activation function: Hardlim 11

Activation function: Purelin 12

Activation function: Logsig 12

5.4 Associative Memory + Classifier 13

Activation function: Hardlim 13

Activation function: Purelin 13

Activation function: Logsig 14

5.5. Patternnet 15

Graphic User Interface (GUI) **16**

Conclusions **17**

1. Introduction

This practical work has the purpose of being an introduction to neural networks (NN): how they work, how to implement and train them using a high-level language like Matlab. For that, the problem addressed was the recognition of characters drawn by humans. However, the main difficulty of this assignment was training the neural networks, since we had to understand how training functions work and in which case they perform better.

In this work we implemented and tested neural networks of one and two layers that work as classifiers and simpler neural networks (associative memory and binary perceptron) implemented in a way they work as filters. This report has the detailed steps of how we proceeded, so that the reader can implement the same architectures.

We start this report by explaining the problem's architecture and how we implemented it. After that, we present all the tests done, the way we got those results and how the NN's performed based on their hit ratio at character recognition. We were able to achieve a maximum hit ratio of 98% using the classifier and the associative memory + classifier, as shown further.

Furthermore, we also created a graphical interface so everyone is able to test our best trained networks and analyze their performance.

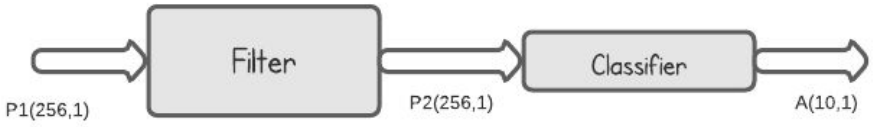
Lastly, we make an overview of what we learned with this project.

2. Neural networks structure

Below are the two architectures that we implemented in order to achieve the best possible in the problem of OCR.

In the first one we start by trying to improve the input character representation quality with a filter. That filter can consist of an associative memory - one single layer, linear activation functions and no bias - or a binary perceptron with 256 hardlim neurons. It's used to get characters more similar to those with which the machine will compare - P2. After that we use P2 to train the classifier, a neural network that can have one or two layers and different activation and training functions and learning algorithms. Each character is represented by a 16x16 matrix, later reshaped into a 256x1 matrix. So the input size of the associative memory, binary perceptron and classifier is 256 in order to match the character pixels number.

To train the filter we need to define the target matrix. T represents the expected output from the given matrix input. In this case, as we draw 1000 characters, T is a 256x1000 matrix, with each column representing a character.



The classifier target (T) matrix is defined by a 10x1000 matrix. Each column represents the expected result (correct answer) times the number of inputs.

In the second architecture only one NN (classifier) is used to classify the input.

3. Implementation

As referred in the previous page the optical character recognition system can be built using different arquitectures. Below we describe how the implementation was done and in which ideas it was based from, so anyone is able to follow the same steps.

3.1 Associative memory filter

The associative memory is a neural network composed by a **single layer**, **without bias** and with **linear activation** functions (meaning that the output from the NN, after the activation function, won't change). In this case we can implement the neural network directly with the following formulas, without needing to resort to the Matlab NN Toolbox:

$$Wp = Target * pinv(P)$$

The function pinv refers to Moore-Penrose pseudoinverse, used to calculate the inverse matrix of the whole input P. After calculating Wp, the weight matrix of the trained NN, to get the output (P2) we just multiply the input (P1) by the weights.

$$P2 = Wp * P1$$

3.2. Perceptron as filter

A different way to correct the input can be done by training a perceptron. This type of NN uses a binary activation function and uses the perceptron learning rule. The following code shows how it can be easily implemented in Matlab.

```
pFilter = perceptron;
pFilter = configure(pFilter,P,T);
pFilter.trainFcn = 'trainc';
pFilter.adaptFcn = 'learnp';
```

The function *configure* allows us to set the perceptron configuration automatically just by giving the input and target matrix's size.

In both implementations, associative memory and perceptron, the target matrix is given by the array of the perfect digits times the size of the input.

3.3. Classifier

The most important part is the classifier. This component is constituted by a neural network of **one or two layers** (we will show how to implement both), with bias and different types of activation functions (hardlim, purelin and sigmoid). In order to achieve the NN max performance, we used several training functions that we will talk about below in the training topic.

This is how we implemented the neural network with **1 layer** and 10 output neurons:

```
%Creating the classifier
net = network(1,1); %netwo
net.inputs{1}.size = 256;
net.layers{1}.size = 10; %

%connect input, bias and o
net.inputConnect(1) = 1;
net.biasConnect(1) = 1;
net.outputConnect(1) = 1;
```

```
%Initialization of
W = rand(10,256);
b = rand(10,1);
net.IW{1,1} = W;
net.b{1,1} = b;
```

And the implementation with **2 layers**: one hidden layer with 15 neurons and one output one with 10 neurons:

```
net = network(1,2); %network wi
net.inputs{1}.size = 256; %laye
net.layers{1}.size = 15; %layer
net.layers{2}.size = 10; %layer

%connect input, bias, layers an
net.inputConnect = [1; 0];
net.biasConnect = [1 ; 1];
net.layerConnect = [0 0 ; 1 0];
net.outputConnect = [0 1];
```

```
net.IW{1,1} = rand(15,256);
net.b{1,1} = rand(15,1);
net.LW{2,1} = rand(10,15);
net.b{2,1} = rand(10,1);
```

In this case, the target matrix is given by the representation of the expected number for each output times the number of inputs.

3.3. Patternnet

In order to build a pattern recognition neural network we defined the number of neurons of the neural network, the training function and the performance function.

```
net=patternnet(10);
net.trainFcn='trainscg';
net.performFcn = 'mse';
```

4. Training

In order to train the neural networks, we created an input of **1000 digits**. Each group member drew 500 using the mpaper.m function and then we concatenated them in a matrix P (input).

Dataset Division:

In order to prevent the training overfitting and to increase the hit ratio, we used the matlab function *divideFcn* to divide the dataset 80% for training, 10% for testing and 10% for validation as shown:

```
%divides dataset into training, vali
net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:800;
net.divideParam.valInd = 801:900;
net.divideParam.testInd = 901:1000;
```

Our object was to achieve a trained NN with the most hit ratio. As so, we did a lot of training by using different learning functions which we will further show the results. This are the independent variables of our study:

Activation functions tested:

```
net.layers{1}.transferFcn = 'hardlim'; %binary
net.layers{1}.transferFcn = 'purelin'; %linear
net.layers{1}.transferFcn = 'logsig'; %sigmoidal
```

Increment learning functions tested:

```
net.adaptFcn = 'learnp'; %perceptron rule
net.adaptFcn = 'learnpn'; %normalized perceptron rule
net.adaptFcn = 'learngd'; %gradient rule
net.adaptFcn = 'learngdm'; %gradient rule improved with momentum
net.adaptFcn = 'learnh'; %hebb rule
net.adaptFcn = 'learnhd'; %hebb rule with decaying weight
net.adaptFcn = 'learnwh'; %Widrow-Hoff learning rule
```

Note: given the nature of *learnp* and *learnpn*, these learning functions are only used with the binary activation function (*hardlim*).

Batch learning functions tested:

```
net.trainFcn = 'traingd'; %gradient descent
net.trainFcn = 'traingda'; %gradient descent with adapt
net.trainFcn = 'traingdm'; %gradient with moment
net.trainFcn = 'trainlm'; %Levenberg-Marquardt
net.trainFcn = 'trainscg'; %scaled conjugate gradient
```

Other variables:

In order to learn more about the neural networks configuration, we also tested the impact of other variables such the learning rate and the *performFnc.* We'll talk about what we learned in the conclusions.

```
net.performParam.lr = 0.5; % learning rate
net.trainParam.epochs = 1000; % maximum epochs
net.trainParam.show = 35; % show
net.trainParam.goal = 1e-6; % goal=objective
net.performFcn = 'sse'; % criterion
%net.performFcn = 'mse'; % criterion
```

5. Results and Discussion

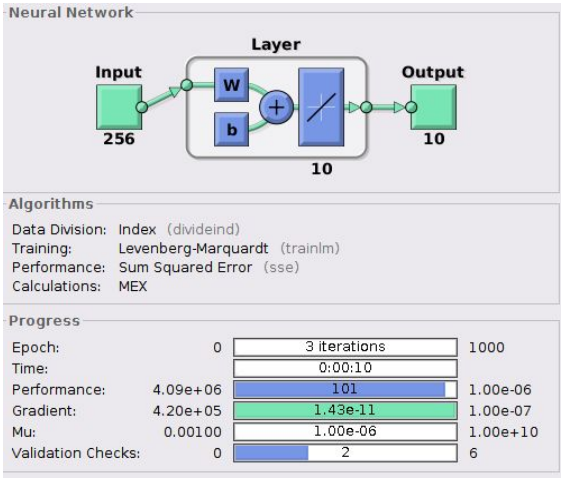
In this section we present the results that we got for each architecture and activation function. Since there are many learning functions we will only put here the ones that gave us the best hit ratios and discuss the results.

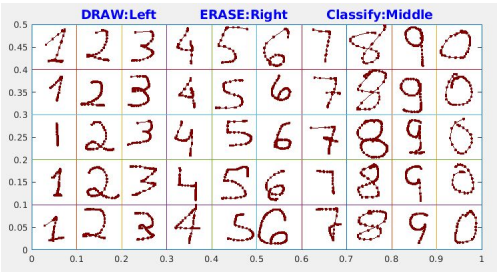
To test we used a dataset composed of 50 numbers (10 of each type). We tried to make them more or less distinct, for **a better generalization**.

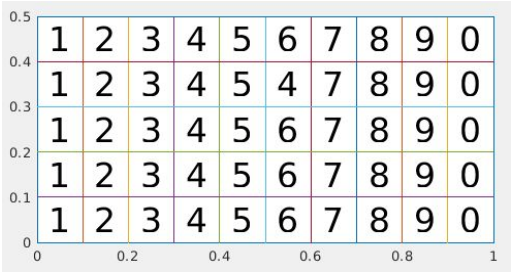
Best Results

- **98% hit ratio : Purelin + Levenberg-Marquardt**

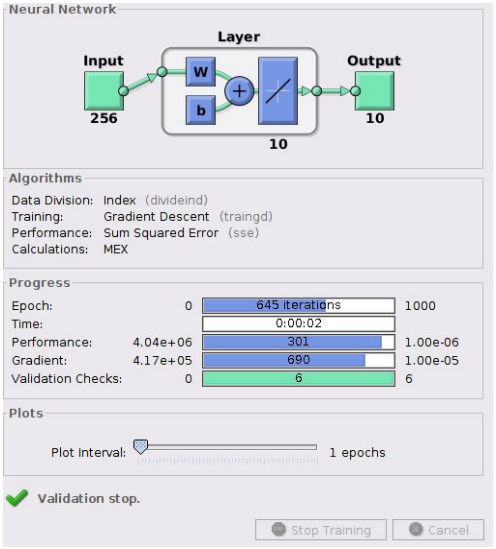
Our best result just using the classifier is a **98%** hit ratio. In order to train the network he used the linear activation function and the Levenberg-Marquardt learning algorithm (batch training).

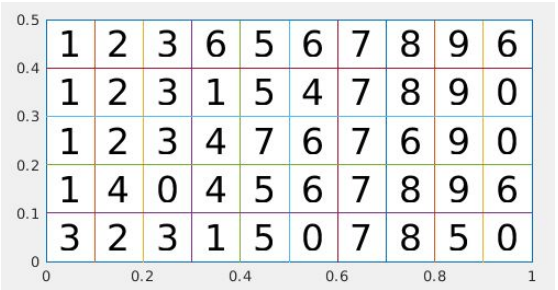


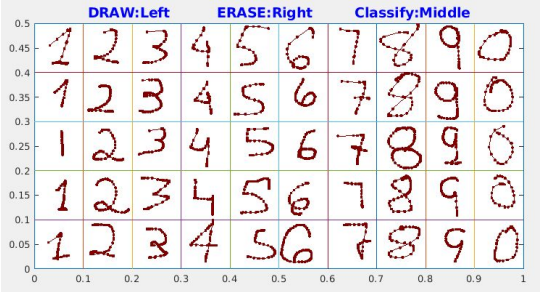




- **96% hit ratio : Purelin + Gradient descent with adaptive learning rate**







5.1. Classifier (one layer)

Activation function: Hardlim

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learnp	-	82%

At the first time we worked with the binary activation function, along with the method to create single layer NN's that work as classifiers explained above, the results weren't great. The output, to any character, was the same.

So we tried an alternative. Thereby, for the hardlim activation function we used a layer of perceptrons, built with the *configure* function, and we were able to achieve a **82% hit ratio**.

Activation function: Purelin

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	4.04e+06	2%
	learngdm	4.22e+06	2%
	learnh	4.22e+06	10%
	learnhd	4.22e+06	8%
	learnwh	4.22e+06	10%
trainr	learngd	4.11e+06	8%
	learngdm	3.99e+06	6%
	learnh	4.13e+06	14%
	learnhd	4.07e+06	10%
	learnwh	4.11e+06	10%
trainb	traingd	4.04e+06	8%
	traingda	4.04e+06 (353)	96% (86%)
	traingdm	4.36e+06	10%
	trainlm	4.09e+06 (101)	98%
	trainscg	4.04e+06 (102)	94%

With the purelin activation functions we obtained some of the **best values of this work**. The batch learning functions performed a better job training the algorithms. However, the incremental

training appeared to give bad results. Still, using random training seems to perform a little better than cyclic training.

In the best results there is a performance value in brackets much smaller than the one next to it. That was the performance value we obtained when we recreated the same NN at the end of work to save for use with the GUI. We didn't understand the reason for this difference, but we added these values to the table as they could be important.

Activation function: Logsig

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	7.20e+03	10%
	learngdm	7.20e+03	8%
	learnh	7.20e+03	8%
	learnhd	7.20e+03	4%
	learnwh	7.20e+03	2%
trainr	learngd	7.20e+03	10%
	learngdm	7.20e+03	20%
	learnh	7.20e+03	12%
	learnhd	7.20e+03	12%
	learnwh	7.20e+03	10%
trainb	traingd	7.20e+03	14%
	traingda	7.20e+03	10%
	traingdm	7.20e+03	12%
	trainlm	7.20e+03	52%
	trainscg	7.20e+03	22%

With sigmoidal activation function, at least in batch training, the results seem to be worse than with the linear one. The best performance was **52% hit ratio**.

5.2. Classifier (two layers)

For the classifier with 2 layers we tried the sigmoidal function in the first layer and the linear function for the second.

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	5.55e+05	10%
	learngdm	4,54e+05	10%
	learnh	4,58e+05	10%
	learnhd	5,22e+05	10%
	learnwh	5,05e+05	10%
trainr	learngd	5,12e+05	10%
	learngdm	5,21e+05	10%
	learnh	5.04e+05	10%
	learnhd	5.01e+05	10%
	learnwh	5.06e+05	10%
trainb	traingd	4.84e+05	10%
	traingda	4.67e+05	10%
	traingdm	5.65e+05	10%
	trainlm	5.55e+05	62%
	trainscg	5.13e+05	12%

The network with two layers performed worse than the one with only one. We were able to reach a **62% hit ratio** with Levenberg-Marquardt training function. The rest of the tests had low performance.

5.3. Perceptron + Classifier

Activation function: Hardlim

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learnp	-	62%

The network with hardlim activation function and with a perceptron as filter has worse results than the classifier alone. However, using the perceptron, this network was able to get **62% right hits**.

Activation function: Purelin

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	8.75e+06	10%
	learngdm	8.95e+06	4%
	learnh	9.30e+06	8%
	learnhd	8.86e+06	8%
	learnwh	8.53e+06	18%
trainr	learngd	9.60e+06	12%
	learngdm	9.18e+06	10%
	learnh	9.33e+06	2%
	learnhd	8.99e+06	6%
	learnwh	8.91e+06	16%
trainb	traingd	9.11e+06	8%
	traingda	9.66e-07	50%
	traingdm	8.30e+06	8%
	trainlm	3.03e-08	54%
	trainscg	4.12e-08	50%

Just like the last point, when comparing this architecture with just the classifier, it seems to perform quietly worse, but the best trained functions were still able to get around **50%** hit ratio.

Activation function: Logsig

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	7.20e+03	0%
	learngdm	7.20e+03	20%
	learnh	7.20e+03	6%
	learnhd	7.20e+03	2%
	learnwh	7.20e+03	0%
	learngd	7.20e+03	4%
	learngdm	7.20e+03	4%

trainr	learnh	7.20e+03	14%
	learnhd	7.20e+03	2%
	learnwh	7.20e+03	0%
trainb	traingd	7.20e+03	4%
	traingda	7.20e+03	10%
	traingdm	7.20e+03	10%
	trainlm	640	22%
	trainscg	5.04e+03	8%

The sigmoid function with the perceptron as a filter has one of the worst performances of this work. It was not able to have a good performance in any of the training functions.

5.4 Associative Memory + Classifier

Activation function: Hardlim

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learnp	-	78%

The perceptron with associative memory has a filter has a similar performance to the classifier alone, so it doesn't seem to improve the performance.

Activation function: Purelin

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	8.52e+06	6%
	learngdm	9.02e+06	18%
	learnh	9.10e+06	14%
	learnhd	8.55e+06	2%
	learnwh	8.45e+06	8%
trainr	learngd	8.68e+06	10%
	learngdm	8.61e+06	6%
	learnh	8.96e+06	6%

	learnhd	8.62e+06	18%
	learnwh	8.62e+06	12%
trainb	traingd	4.28e+06	10%
	traingda	4.11e+06 (160)	80% (98%)
	traingdm	4.15e+06	10%
	trainlm	4.13e+06 (116)	98%
	trainscg	4.23e+06 (116)	98%

On the other hand, using the classifier with linear function together with a filter composed by associative memory helped the classifier to reach higher performances than without the filter.

The discrepancy in the values has the same reason as explained in the analysis of the results of the NN composed by only a single layer classifier with linear activation function..

Activation function: Logsig

Training Function	Learning Algorithm	Performance	Hit ratio
trainc	learngd	7.20e+03	6%
	learngdm	7.20e+03	18%
	learnh	7.20e+03	8%
	learnhd	7.20e+03	2%
	learnwh	4.10e+03	6%
trainr	learngd	7.20e+03	14%
	learngdm	7.20e+03	16%
	learnh	7.20e+03	12%
	learnhd	7.20e+03	6%
	learnwh	7.20e+03	10%
trainb	traingd	5.91e+03	4%
	traingda	6.58e+03	6%
	traingdm	7.20e+03	4%
	trainlm	7.20e+03	32%
	trainscg	3.28e+03	10%

The associative memory also improved the performance of the classifier, however most of the training functions do not recognize many digits well.

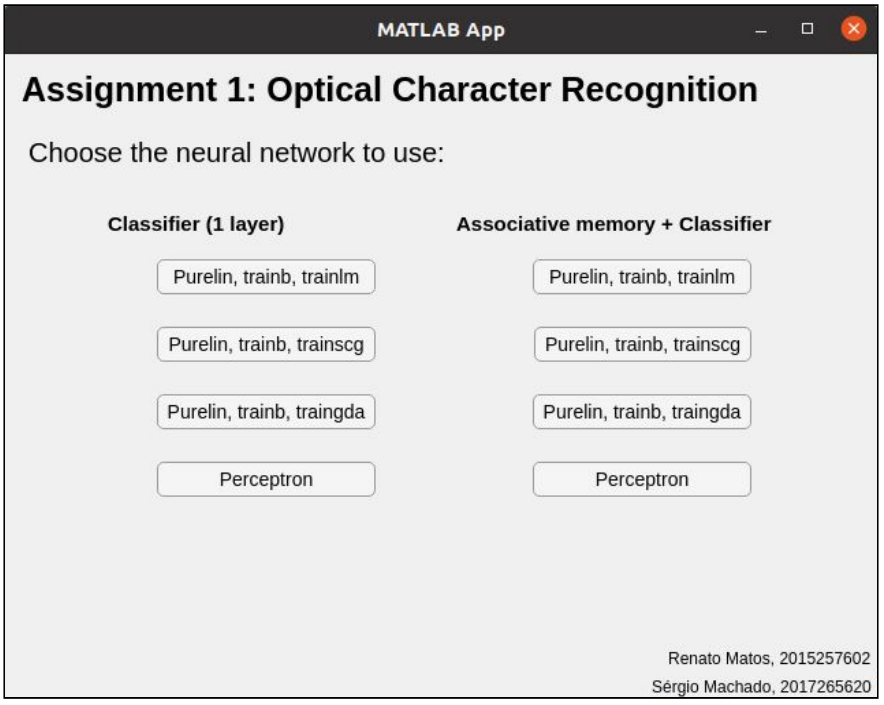
5.5. Patternnet

Learning Algorithm	Performance	Hit ratio
traingd	0.0634	40%
traingda	5.54e-06	82%
traingdm	0.0651	32%
trainlm	7.86e-07	82%
trainscg	8.43e-07	84%

The pattern recognition neural network proved to be an excellent option to use instead of the classifier. It doesn't have a higher hit ratio like others but has a good performance.

The use of this type of NN was a last-minute decision, so it was not added to use with the GUI. However, given the interesting results, it was worth experimenting.

6. Graphic User Interface (GUI)



As requested we also built a graphic user interface (GUI) so that a user can test the trained networks that got better results. The selected ones are the eight NN's seen in the print of the GUI.

To use the GUI it's just needed to access **GUI.mlapp** in the current folder tree within Matlab. By clicking, the App Designer will open and the option to **run** the app will appear on top.

Within the app, you just have to click on the button corresponding to the desired network and the *mpaper* will appear to draw the characters to be analysed by the network.

7. Conclusions

After testing different training and learning functions for each architecture we were able to reach a max performance of **98% hit ratio** using only the classifier and the associative memory + classifier. To achieve that number we used the linear activation function and the Levenberg-Marquardt learning function in both arquitectures. We also got other excellent results using only the classifier: **96%** and **94% hit ratio** with linear activation function and gradient descent with adaptive learning rate (traingda) and scaled conjugate gradient (traincsg), respectively.

When using the perceptron as a filter there is a clear performance decrease to nearly half of the performance with just the classifier. However, when using the associative memory, we notice a slight increase on the hit ratio.

Making a lot of tests made us understand that, overall, the batch training gives higher results and takes a lot less time to train since it only updates the neuron's weights at the end of the epoch. Despite the increment training resulting in worst trained neural networks, giving the training samples in a random way (trainr) seems to have an impact in getting better NN than cyclic training. Also, the linear activation function was the one that provided us the best results.

The binary function, used as perceptron, also obtained relatively good results. That was a very extremely good surprise since our first implementation of NN's with binary activation function had poor results.

We were able to achieve trained neural networks with a good generalization capacity since the test hit ratio results were based on the hit ratio of a data set with digits drawn in a different way than the digits from the training set.

We realized that if we made the training dataset with similar data the classification system would perform worse since it would have less generalization capacity and a chance of overfitting. Because of that we tried to make a data set with differently drawn digits.