Index

[1. Data 2](#_Toc88064574)

[2. Implementation – Neural Network Architectures 2](#_Toc88064575)

[3. Results 3](#_Toc88064576)

[3.1 Filter and Classifier Results 3](#_Toc88064577)

[3.2 Classifier 4](#_Toc88064578)

[3.3 Learning Rate 4](#_Toc88064579)

[3.4 Dataset 4](#_Toc88064580)

[3.4.1 Training 4](#_Toc88064581)

[3.4.2 Validation and Test 4](#_Toc88064582)

[3.4.3 Size 4](#_Toc88064583)

[3.4.4 Filter 4](#_Toc88064584)

[4. Conclusions 4](#_Toc88064585)

# Data

Each character is stored in a 16 by 16 matrix of binary values generated with the function *mpaper.m* that allows the user to draw 50 characters with the mouse. After creating the matrix, it is then reshaped into a 256-vector column that will be the input of 50 examples. By concatenating various 50 digit matrixes created previously we created datasets of two sizes, 500 digits and 1010 digits.

To achieve a better result, we attempted to have our digits drawn by different people, so that the data’s variation could improve and generalize better and let the testing results be more legitimate.

In the file *PerfectArial* there is another dataset that contains the perfect writing of the ten digits, and it is to be used as target for the filters.

The target dataset for the Classifier stage is a 10 by 10 identity matrix concatenated with itself as many times as needed. Each column represents a digit, with index one being digit one, and index ten being digit zero. The size must be adjusted so that both P and T matrices represent the same number of digits.

# Implementation – Neural Network Architectures

The task required several different architectures:

* Filter + Classifier
* **Associative Memory** as filter (**AM**): the module Filter takes the input vector P1 (256,1) and it “corrects” the data; then it returns the output vector P2 (256,1) that will be a better representation of the character. the target used is *PerfectArial.mat* matrix.
* **Binary Perceptron** as filter: A binary perceptron acts like the filter and it has 256 neurons that has the same target of AM filtering with learning algorithm *learnp* and incremental training function *trainc*;
* **No filter:** directly send a dataset as input to the classifier;

In both filters the generated output serves as input to a Classifier;

* Classifier Layers
* **One Layer**: Classifier with 1 layer (output layer) that receives an input vector P1 (256,1) and provide the output vector A (10,1). The output is set to 10 neurons.

The initialization of W and b parameters was random.

* **Two Layers**: Classifier with 2 layers (hidden layer, output layer) that receives an input vector P1(256,1) and provide the output **A** (10,1). The number of neurons of the hidden were defined by us in the range [10; 100] to reach the best classifier. We settled for 20.

The initialization of W and b parameters was chosen by the default initialization (pattern recognition toolbox’s default).

* Some of the training functions suggested for this project were *learnp* – *learnp* – perceptron rule, *learnpn* – normalized perceptron rule, *learngd* – gradient rule, *learngdm* – gradient rule improved with momentum, *learnh* – *hebb* rule (historical), *learnhd*- *hebb* rule with decaying weight, *learnwh* Widrow-Hoff learning rule for incremental training; and *traingda* gradient descent with adaptive leaning rate, *traingdm* gradient with moment, *trainlm* Levenberg- Marquardt, *trainscg* scaled conjugate gradient for batch training.
* The activation functions suggested for the task were *hardlim*, linear, sigmoid and softmax.

The results are normalized ( although hardlim does not need it) so that the output is a binary matrix in which the columns represent the classified number for each example.

For what concern the training parameters, they were left as the default specifications in the pdf:

* **Learning Rate**: it was set to 0.1 to best adapt to the data
* **Epochs**: it was set to 1000 so that the models have enough time to converge to a good solution
* **Goal**: it was set to
* **Perform Function**: sum squared error for the one layer and cross-entropy error for the two layers

# Results

We tested and compared several approaches used in the work. They differentiated by:

* **Architecture**: Filter and Classifier or just Classifier
* **Filters**: Perceptron or AM Filter
* **Number of Layers**: 1 or 2 layers
* **Activation function**: linear, hardlim, sigmoid, softmax
* **Train function**: *“trainr”, “trainlm”, “trainscg”, “trainc”*
* **Learning algorithm**: *“learnp”, “learngdm”, “learnh”*

For our metric, the accuracy of classification was considered in the form of a percentage. We used a base input of 50 digits in the [1,2,…,9,0] format, and compared it to a pre-initialized matrix with the same format. This would then yield a percentage representing the accuracy of the classification. This method is only valid for this format but with such a large input we believe it functions as an accurate metric.

Before deciding on the training functions, we researched that:

trainlm - best for regression

trainscg - best for classification and unit column vector targets

trainrp - best for huge datasets

## 3.1 Filter Results

|  |  |  |  |
| --- | --- | --- | --- |
| Filter + activation | Training | Learning | % suceess |
| AM + Hardlim | *Trainr* | *Learnp* | 93 |
| AM + Hardlim | *Trainc* | *Learnp* | 89 |
| AM + logsig | *Trainc* | *Learnp* | 91 |
| AM + logsig | *Trainr* | *learngdm* | 98 |
| AM + Hardlim | *Trainc* | *learngdm* | 88 |
| AM + logsig | *Trainc* | *learngdm* | 98 |
| AM + Hardlim | *Trainr* | *learnh* | 93 |
| AM + logsig | *Trainr* | *learnh* | 92 |
| AM + Hardlim | *Trainc* | *learnh* | 92 |
| AM + logsig | *Trainc* | *learnh* | 92 |
| AM + Purelin | *Trainlm* |  | 97 |
| AM + logsig | *Trainlm* |  | 97 |
| AM + Purelin | *Trainscg* |  | 98 |
| AM + logsig | *Trainscg* |  | 98 |
|  |  |  |  |
| perc + Hardlim | *Trainr* | *Learnp* | 100 |
| perc + Hardlim | *Trainc* | *Learnp* | 100 |
| perc + logsig | *Trainc* | *Learnp* | 100 |
| perc + logsig | *Trainr* | *learngdm* | 100 |
| perc + Hardlim | *Trainc* | *learngdm* | 100 |
| perc + logsig | *Trainc* | *learngdm* | 100 |
| perc + Hardlim | *Trainr* | *learnh* | 100 |
| perc + logsig | *Trainr* | *learnh* | 100 |
| perc + Hardlim | *Trainc* | *learnh* | 100 |
| perc + logsig | *Trainc* | *learnh* | 100 |
| perc + Purelin | *Trainlm* |  | 100 |
| perc + logsig | *Trainlm* |  | 100 |
| perc + Purelin | *Trainscg* |  | 100 |
| perc + logsig | *Trainscg* |  | 100 |

From the results we can see that both filters had nearly100% precision in training. This is the reason that makes us believe that filters degrade the generalization capability of the classifiers we will talk about further on. The filters seem to be overfitted. Neither is efficient in guessing numbers during the test phase, with accuracies near the 10%. This seems to be further proved by the fact that AM is slightly better compared to perceptron at guessing numbers, when perceptron had better training results.

About the Activation Functions: *Hardlim* gave the worst results of three while sigmoid and linear were similar at this time.

We found that the activation functions were the biggest influence, and that the training and learning functions had a lesser impact on the filter results.

## 3.2 Classifier Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Classifier | Filter | Training | Learning | % suceess |
| Hardlim | No filter | trainscg | learngdm | 11 |
| Hardlim | AM | trainscg | learngdm | 8 |
| Hardlim | Perc | trainscg | learngdm | 5 |
| Hardlim + softmax | No filter | trainscg | learngdm | 13 |
|  |  |  |  |  |
| Purelin | No filter | trainscg | learngdm | 18 |
| Purelin | AM | trainscg | learngdm | 13 |
| Purelin | Perc | trainscg | learngdm | 13 |
| Purelin | No filter | *trainlm* | *learnh* | 18 |
| Purelin | No filter | *trainlm* | learnnp | 18 |
| Purelin + Purelin | No filter | trainscg | learngdm | 55 |
| Purelin + hardlim | No filter | trainscg | learngdm | 11 |
| Purelin + logsig | No filter | trainscg | learngdm | 18 |
| Purelin + softmax | No filter | trainscg | learngdm | 32 |
|  |  |  |  |  |
| Softmax | No filter | trainscg | learngdm | 8 |
|  |  |  |  |  |
| Logsig | AM | trainscg | learngdm | 25 |
| Logsig | Perc | trainscg | learngdm | 22 |
| Logsig | No filter | trainscg | learngdm | 76 |
| Logsig | No filter | *trainlm* | learngdm | 73 |
| Logsig | No filter | *trainr* | learngdm | 12 |
| Logsig | No filter | *trainc* | learngdm | 15 |
| Logsig | No filter | trainscg | *learnp* | 63 |
| Logsig | No filter | trainscg | *learnh* | 71 |
| Logsig + hardlim | No filter | trainscg | learngdm | 12 |
| Logsig + Purelin | No filter | trainscg | learngdm | 60 |
| Logsig + logsig | No filter | trainscg | learngdm | 23 |
| Logsig + softmax | No filter | trainscg | learngdm | 82 |
| Logsig + softmax | AM | trainscg | learngdm | 34 |
| Logsig + softmax | Perc | trainscg | learngdm | 24 |

We simulated with both one and two layer classifiers, with/without filters, and with different train and learn functions. Our tests consisted of the same 5 inputs made of 50 digits each, in the row[1…9,0] format. At first we ran each input 3 times but the results were the same, so we began running each input just once per test.

We arrived to the conclusion that pre-filters cause overfitting and are counterproductive, as all the classifiers with filters underperform. We tested the filters with both one and two layer classifiers, both yielding similar results of overfitting.

The best accuracy was reached with a two-layer classifier, with no filter, and with a *Logsig*-*Softmax* combo, which reached 85% accuracy. The difference in accuracy between a one and two layer Logsig was negligible unless softmax was used. *Softmax* perform a logistic processing of the input data so that the output of the network is normalized to a probability distribution over a predicted output classes and this normalization helped with the later normalization to a binary matrix. It worked best as a Post-Filter; The average difference of a Logsig one layer classifier and a Logsig-Softmax two layer was about 10-15%.

By considering the results as a whole, we can see that the average precision for the architecture with two layers is higher; a slight improvement can be seen by the value obtained with *Softmax* as the second activation function although it doesn’t work as well in as first activation function. *Purelin* provided a bellow average result, while hardlim was consistent in being the worst, and *Logsig* gave the best result as first activation function in all of the architecture variations.

Interestingly, a dual purelin classifier fared well enough to be noticeable, and after tests with a logsig with a second purelin layer, we believe that purelin fared somewhat well as a post processing layer, similar to softmax but not as good. Logsig as a second layer fared as bad as hardlim, so in our simulations logsig was of no use as a post processing layer.

Considering all the training function used, *trainscg* seems to be the best choice for pattern recognition networks known for the speed and low memory requirements, although trainlm is a faster training function according to specifications and also fared decently. Trainc and Trainr both destroyed the classifiers performance, and both are incremental, so it leads us to believe batch functions are better for training classifiers. The learning functions we tested didn’t vary in effect by a big margin. While we found learngdm would be better overall with previous research, our tests show us that while it is indeed better, it’s still not a big difference at 5%.

## 

## 3.3 Dataset

### 3.3.1 Training, Validation and Test

The training data seems to give a better accuracy than validation and testing set. This is due to overfitting the model, which only learns how to classify the training data and lack generalization capability, therefor simulating worse for the other datasets, validation and test.

Our sets were split in a segmented way so as to assure all digits were equally trained, avoiding cases like for example training all digits but ‘6’, and therefor causing the digit 6 to be less accurate in classification.

We also followed the common procedure of splitting the training, validation and test in a 70-15-15 format for classifiers and 70-30 for filters. We used matrices drawn by several people so as to allow a more efficient training of classifiers.

*Trainscg* was again tested against the other training functions, and proved to be the best, although the biggest improvements came from the activation functions.

### 3.3.2 Size

To try and avoid overfitting, a large amount of data was used: two datasets were created, with 500 and 1010 digits. While the initial tests were done using the 500 digit dataset, after creating and switching to the 1010 dataset the overall accuracy increased by 10%, proving that a larger and more diverse dataset allows for further improvement. This then brings into the table the issue of the no lunch theorem. This means that the size of the dataset needs to be set while having in mind whether the increase in size, and therefor training time and simulation (less runtime performance), is worth the increase in accuracy (result). We are not sure what the optimal size would be, but we found that the increase in runtime from 500 to 1010 digits was more than double in some cases, with the 500 digit dataset training classifiers within one minute, while the 1010 digit took almost 5 minutes.

### 3.3.3 Filter

The use of filter method gave low accuracy results for the test while providing high results for the training since the filter contributes to overfitting: it makes the data uniform so that the model learns just those specifics patterns that filtering method produced.

# Conclusions

Since the filter caused overfitting and therefor caused the models to underperform on new datasets, we can say that the filter didn’t improve OCR’s performance. We stopped using filters after initial tests showed us that they were counterproductive to all classifiers.

Both classifiers with 1 layer and 2 layer classified the numbers well when combining Logsig and Softmax. with others efficient validation and training functions. The activation functions and dataset size were the main factors of improvement. Dual Purelin was only adequate.

Training functions also played a crucial role as trainscg outperformed the others quite consistently.