Artificial Neural Networks reports

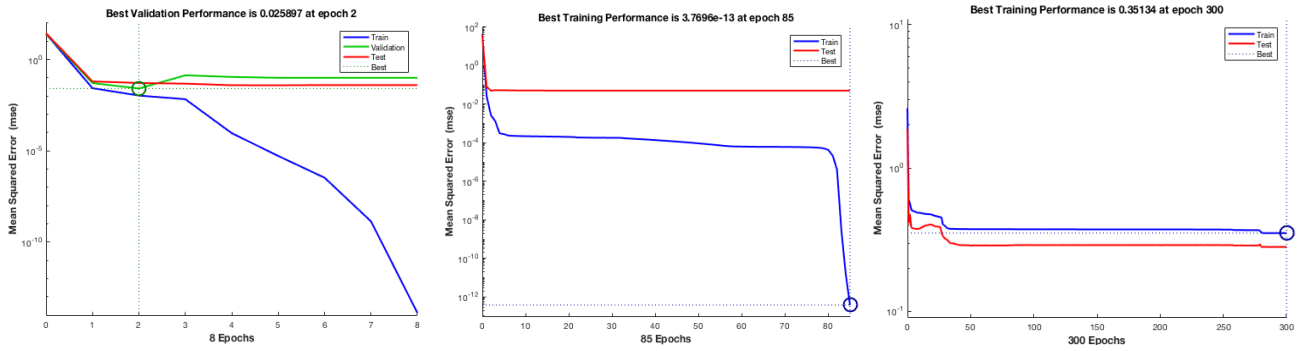
Mauro Paradela del Río - r0696150

1 Lab 1

1.1 Algorithm comparison

		gd					**gda**					**cgf**		
N	R_t	MSE_t	Epch	T(s)	N	R_t	MSE_t	Epch	T(s)	N	R_t	MSE_t	Epch	T(s)
10	0.06	0.49	771	2.64	10	0.17	0.51	79	0.28	10	0.15	0.50	11	0.10
20	0.31	0.37	849	2.82	20	0.25	0.48	79	0.53	20	0.42	0.38	17	0.31
40	0.32	0.35	915	3.58	40	0.29	0.54	67	0.58	40	0.65	0.28	30	0.54
80	0.49	0.27	958	4.12	80	0.22	0.88	68	0.59	80	0.69	0.29	44	0.53
100	0.48	0.53	976	4.90	100	0.20	0.90	112	0.51	100	0.68	0.35	37	0.59
120	0.47	0.62	957	7.15	120	0.21	1.15	97	0.47	120	0.60	0.45	38	0.76
		cgp					**bfg**					**lm**		
N	R_t	MSE_t	Epch	T(s)	N	R_t	MSE_t	Epch	T(s)	N	R_t	MSE_t	Epch	T(s)
10	0.17	0.48	12	0.10	10	0.18	0.48	11	0.15	10	0.18	0.44	11	0.07
20	0.33	0.48	15	0.40	20	0.43	0.40	17	0.30	20	0.50	0.38	6	0.10
40	0.60	0.32	25	0.48	40	0.77	0.20	35	0.86	40	0.85	0.13	11	0.18
80	0.67	0.31	35	1.59	80	0.91	0.9	39	2.34	**80**	**.93**	**.07**	**6**	**.31**
100	0.62	0.38	40	0.66	100	0.88	0.12	47	2.55	100	0.90	0.10	4	0.26
120	0.62	0.44	32	0.63	120	0.75	0.28	47	3.59	120	0.81	0.22	5	0.27

Table 1: Performance of different training algorithms. Results are the average ones after twenty runs on the test set. N is the neurons on the hidden layer. R_t and MSE_t stand for regression R-value and for MSE on the test set, respectively. Test size is 15% of the total data. *Epch* stands for epoch of convergence. *T(s)* is time (seconds). Best result is highlighted in bold. The dotted line separates overfitted results



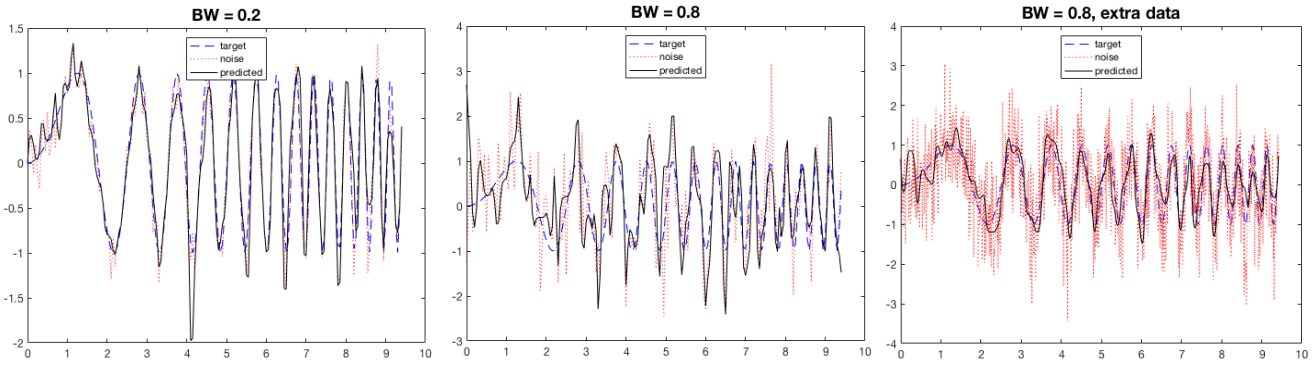
(a) Validation set avoids overfitting (b) Overfit. 120 Neurons (c) Underfit. 5 Neurons

Figure 1: Evolution of test (red), train (blue) and validation (green) set MSE during training. Left figure uses a validation set that stops befor e overfitting. Otherwise, train error keeps sinking after the optima, a s Figure 16b also shows. On the right, with 5 neurons, the training never converges: the net is too simple to learn the function, error never decays. Test error (red) is lower than training error, it underfits.

The goal here is comparing the six algorithms for training Neural Nets on the Matlab toolbox. For this purpose, I have trained a net using each algorithm with different number of hidden units. Then, I have evaluated is performance on the test set, and repeated the process twenty times. Table 1 shows the results of these experiments, averaged for each algorithm and neuron setting. Overall, all algorithms perform best with 80 hidden units: nets with neurons over that threshold are overparametrized, since their MSE on the test set starts to decrease. If there was no validation set, for 100 and 120 neurons the MSE on train set would be way lower than that of test set, as Figure 16b shows. On the other hand, nets with few hidden units do not even learn the model, and can have higher train error than test, as in Figure 16c.

The best algorithm in terms of speed and performance is the *Levenberg-Marquardt* (lm) algorithm: it trains the fastest and it reaches better performance than the other ones under the same configurations. *Gradient descent* is by far the worst, both in speed and accuracy.

1.1.1 Noisy data



(a) $R_t = 0.54$, $MSE_t = 0.53$ (b) $R_t = 0.38$, $MSE_t = 2.25$ (c) $R_t = 0.58$, $MSE_t = 0.82$

Figure 2: Estimation of a function with Gaussian noise, 0 mean, and different σ, or BW. Figures (a) and (b) estimate on 190 data points, figure (c) estimates on 950, 5 times more points. The network contains 80 hidden units, and it was trained using the *lm* algorithm. MSE with noisy data is worse, compared with the same setup on Table 1. Increasing the data points reduces again the error, as (c) shows.

Figure 2 demonstrates that adding noise to the function affects the network estimation. It basically increases its error, no matter its setup. In this case, having more data points makes error go back to its original result: Figure 2c shows it. This result showcases another property of neural networks: they handle noisy data better than other estimators and even better when there is even more data, without increasing the demand for memory (opposed to SVM, where more data points mean bigger Kernel matrices).

1.2 Bayesian Inference

Levenberg-Marquardt						Bayesian Optimization					
N	R_t	MSE_t	MSE_{tr}	Epch	T(s)	N	R_t	MSE_t	MSE_{tr}	Epch	T(s)
50	0.97	0.03	0.81×10^{-2}	8	0.15	50	1.00	10^{-4}	0.94×10^{-7}	150	2.11
100	0.90	0.10	0.29×10^{-2}	6	0.33	100	0.98	0.02	0.50×10^{-2}	106	6.52
150	0.76	0.28	0.01×10^{-2}	5	0.31	150	0.89	0.11	0.22×10^{-4}	33	13.69
300	0.41	1.00	0.27×10^{-2}	3	1.28	300	0.61	0.62	0.19×10^{-7}	27	92.65

Table 2: Comparison between the Levenberg-Marquardt algorithm and Bayesian optimization, using over-parametrized networks. t stands for test set, tr stands for training set. As the hidden units increase, the `trainlm` algorithm starts to overparametrize: validation set makes training finish earlier and error on the test set increases too, whereas *MSE* on the training set remains constant. On the other hand, Bayes prior tries to keep the weights small, so test error increase is not as steep. In exchange, training takes more time, because finding the probabilities of each model takes much time.

Table 2 compares the performance of Bayesian learning against the best performing algorithm from the previous section. The table showcases the main advantage of Bayesian optimization: it minimizes the weights of the network thanks to its prior distribution. Therefore, performance is better when there are too many hidden units. It comes at a price: training is way slower because there is no validation set; it calculates the probability distribution of the prior and the posterior at each level, which takes more time than regular backpropagation. The table also shows a common problem when searching the best parameter configuration: the `trainlm` algorithm reached its peak performance with 50 neurons, but I skipped this setup on Table 1

2 Lab 2

2.1 Hopfield Network

Hopfield networks are recurrent NN with binary valued units that can store pattern according to the principles of associative memory, regarding storage capacity and retrieval. In this section I will illustrate their behaviour using them to retrieve original digits from noisy samples.

First, the network receives an input consisting on the initial attractors. In this case, the attractors are 9 handwritten digits, scanned as 15×14 pixel images. So, the network has 240 features, as many as the attractors. The network can store up to $N/(4 \times log(N))$ patterns, $N = 240$. That result is 10.86, so the net should be able to store all of them without getting spurious patterns.

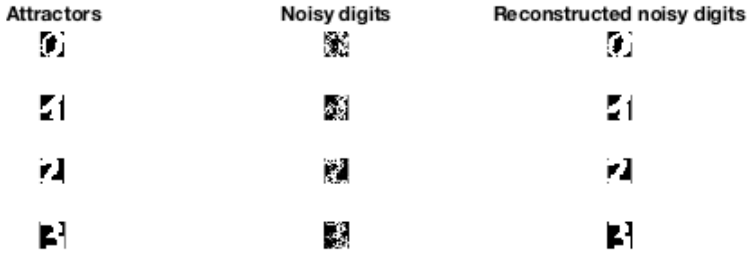
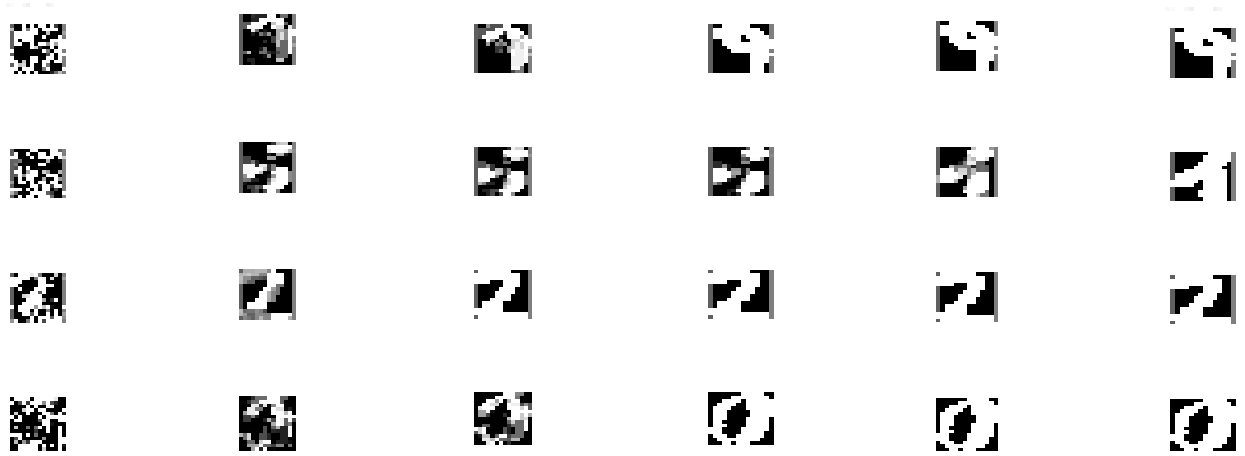


Figure 3: Retrieved digits after 10 iterations, using a noise factor of 1. Reconstruction is perfect, all patterns are retrieved



(a) Nois (b) 10 (c) 20 (d) 30 (e) 40 (f) 50

Figure 4: Retrieval process of the Hopfield network, varying the number of iterations, in the subcaptions. Noise factor is 10. At first the network has not reached any minima, so it delivers noisy digits. As it keeps iterating, it approaches the final state. Here all original attractors are memorized, but noise is so strong that the network can retrieve the wrong pattern.

Indeed, no spurious pattern is retrieved. With low noise, as in Figure 3, reconstruction is perfect even after 10 iterations, and it remains stable afterwards (low *crosstalk*). On Figure 4 noise is way higher. It takes more time to converge, so the digits retrieved during the first 40 iterations still have some noise, equilibrium is only reached after 50 epochs. Furthermore, corruption has pulled the data points too close to the wrong attractors, so the wrong digit is commonly retrieved. But, as dimensionality is so high, all patterns are properly stored, and there is no convergence on spurious, local minima.

2.2 Elman network

Elman networks are a kind of Recurrent Neural Networks, were the output is fed to the input for the following training/simulation step. Therefore, they can learn temporal patterns from data, and they are often used for time series prediction.

2nd layer	1st layer	10	50	100	500
`purelin`	`tansig`	1.05	0.32	0.27	0.38
	`logisg`	0.97	0.92	0.31	0.43

2nd layer	1st layer	10	50	100	500
`tansig`	`tansig`	44.46	1.57	2.81	0.71
	`logisg`	1.35	0.95	0.92	1.26

Table 3: MSE for different combinations of training epochs and training functions. The best architecture seems to be having a `purelin` architecture on the second layer, and a `logsig` on the second. All nets had 50 neurons, and were trained using 500 datapoints. The shown results have been averaged over 20 experiments

I have tested the network on the *Hammersteim system* data using different configurations. First, I have tuned the training set size: I have used 100, 200, 300 and 500 datapoints under the default configurations (50 neurons, 500 training epochs), and they have delivered 0.41, 0.30, 0.22 and 0.07 MSE on the test set, respectively. So, the network performance improves as it has more data points, similarly to other kind of networks.

Increasing the neurons (beyond 50) decreases performance: testing with 50, 100, 150 and 250 I have obtained MSE of 0.14, 1.52, 2.20 and 3.78 respectively. 50 is already good; more than that overparametrizes the network.

I also also have tried different architectures and training epochs, shown in Table 3. As expected, the second layer should use a linear activation function: having non linear activation functions on the last layer pushes prediction values towards 1 and -1. Of the both tested functions, the hyperbolic tangent works best. Increasing epochs improves performance until they exceed 100; particularly on the `purelin` function. This means that the network is overfitting beyond that point.

Finally, I am showing the result of training and testing the network using an optimal configuration in Figure 5.

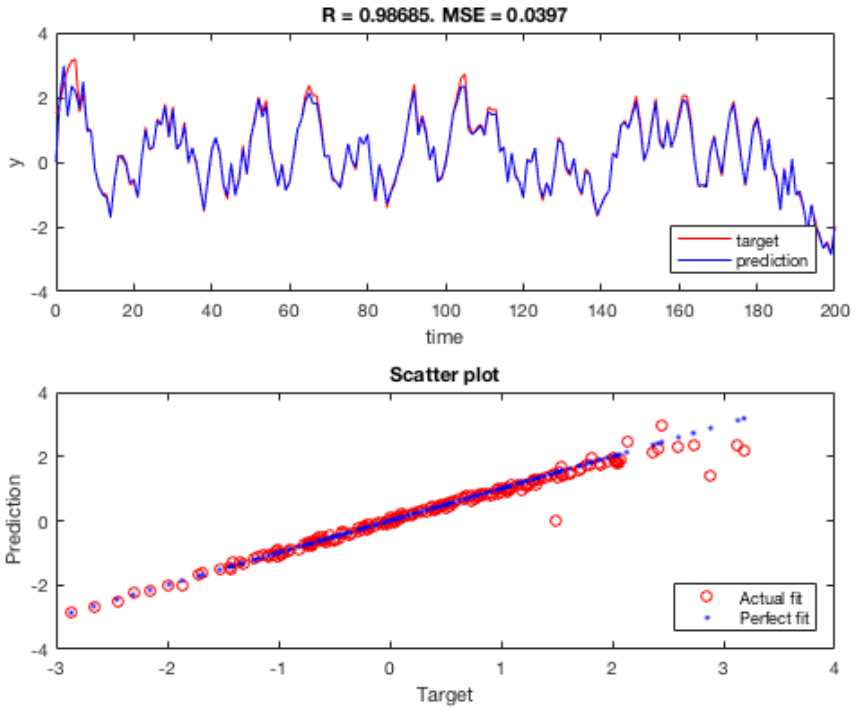
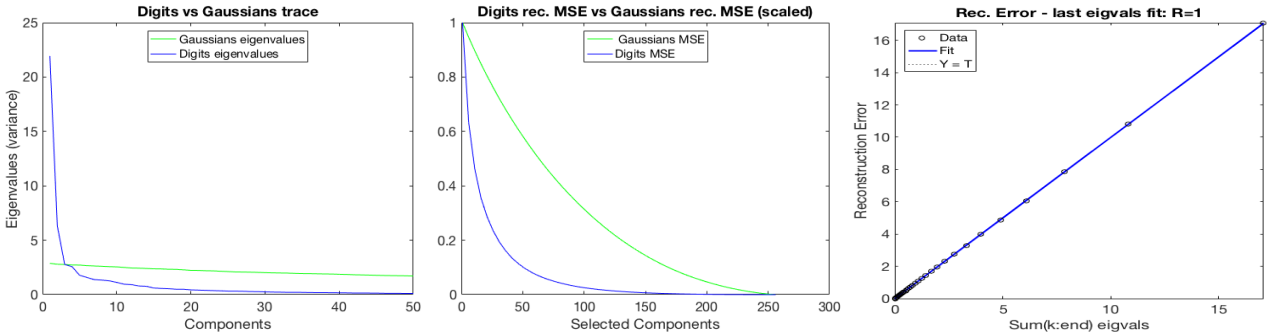


Figure 5: Test set function estimation on the *Hammersteim system*, using the optimal parameter configuration from the previous experiments: 50 hidden units, 100 epochs, an hyperbolic tangent transfer function on the first layer, a linear function on the second and 500 points for training.

3 Lab 3: Unsupervised Learning

3.1 Principal Component Analysis



(a) Eigenvalues by PC (b) Rec. *MSE* by PC (scaled) (c) Lin. fit: MSE and left-out variance

Figure 6: Comparison between a synthetic dataset (green) and the *threes* dataset, using their eigenvalues by PC (left) and their reconstruction error (centre, scaled by their max. value). Right image shows a perfect linear relation between recovery error and the summed variance of all the components but the first k ones.

The goal of this section is to understand the meaning and consequences of using PCA. Figure 6 compares its behaviour on the *threes* dataset and on a random dataset of 256 I.d. Gaussian features. For the *threes* most information lies on the first PC: according to Figure 6a, 50 components explain already 90% of the variance. On the Gaussians, that makes less than 40%. Therefore, 50 components should already deliver a good reconstruction of the digits, as Figure 6b shows: MSE decreases steeply until then, and slowly afterwards. On the other hand, on the Gaussian dataset information is distributed evenly among the components. Both eigenvalues and MSE do not decrease so sharply, there is no optimal transformation of the data such that more information lies on a specific direction.

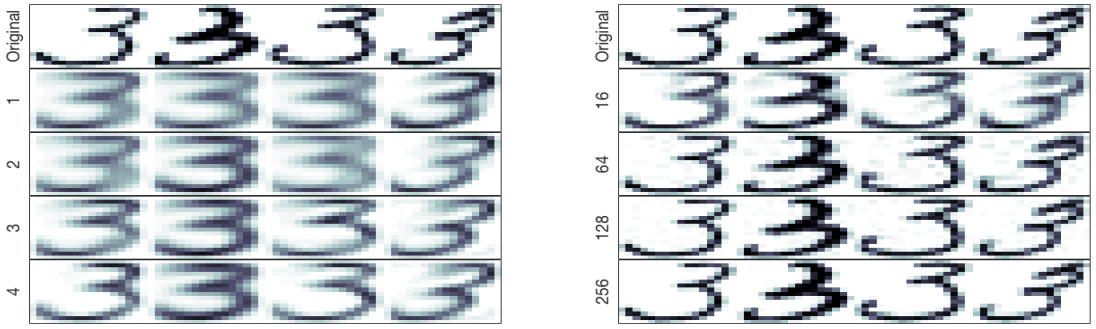


Figure 7: Reconstruction of four different samples (columns) with increasing number of components (rows). Top row of each table contains the original samples. Vertical labels indicate the number of components

Visual reconstruction a sample with few components if much different from the original image, as the images on Figure 7 show. These recovered images seem more similar to the average three. The reason is that the first components model the main structure underlying all the data, the *essence* of the *threes*. As more components are added, reconstruction becomes more accurate. The last components model the noise, the particularities of each data point. Since most variance lies below 50 PC, any reconstruction with more PC delivers almost the same result, there is little margin of improvement. With 256 PC the recovered image is exactly the same, and the *MSE* is 0. There is a mathematical proof: being data $X \in \mathbb{R}^{d \times N}$, with d the input feature space; $V \in \mathbb{R}^{d \times s}$ the eigenvector matrix, s selected components; and $X_s = V^t X$ the data projected on those components. If $s = d$, then $VV^t = I$ (full rank orthogonal matrix). Then, $\hat{X} = VX_s = VV^t X = X$.

Component interpretation each PC models data particularities, that can be used to find patterns on it, and even for clustering. Figure 8 shows a representation of the first and fifth component. Then, I have

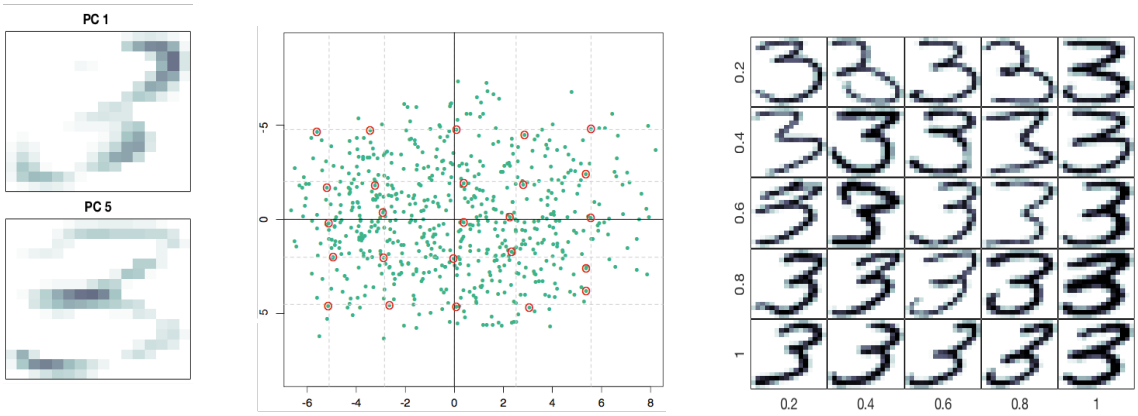
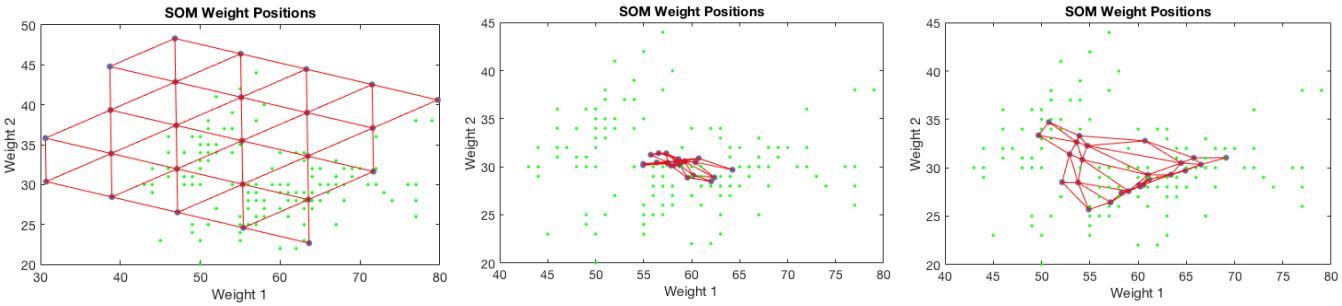


Figure 8: 1st and 5th PC (left), and data points when projected on them (centre). X axis correspond to the fifth, Y to the first. Red dots show the closest data point to a quantile on the projections distribution. These samples are shown on the right image, with their corresponding quantile.

taken the points corresponding to different quantiles of the data, when projected on those components. The results are shown on the right image. PC 1 models cursiveness: threes on the Y axis lean to the left side on the low quantiles (top), and become totally cursive on the bottom rows. The fifth (X axis) models thickness on the horizontal strokes: digits at the highest quantile (quantile) are the thickest.

3.2 Self Organizing Maps



(a) Network initialization. (b) Training after one epoch (c) Final result

Figure 9: Training state evolution of a SOM with 25 neurons, set in a hexagonal grid of 5×5. On the first epoch, many neurons are pulled together into the regions of high variance, since their neighbourhood radius is very large. At later stages, neighbourhood size shrinks and less neurons are pulled from their position.

SOM deliver meaningful representations of the data (vector quantization) using competitive learning to make their neurons learn the data distribution. Basically, as a neuron "wins" over the others for a data point, its weight is updated so it is closer to the input, according to the Konohen rule. The new state depends on both the previous state and the input, and each has a different weight (parameter α). Besides, neurons also pull their neighbours, defined by a radius on some distance metric and grid disposition. Figure 9 showcases the learning process of the SOM. Furthermore, they can also be used for clustering, with as many clusters as neurons. Table 4 compares clustering performance with different configurations, using 3 clusters.

	`linkdist`	`dist`	`boxdist`	`mandist`
Hexagonal	0.7240	0.7240	0.7254	0.7239
Grid	0.7254	0.7233	0.7247	0.7226
Random	0.6817	0.7240	0.7246	0.7240

Table 4: Comparison of different topologies and distances for the iris dataset, using a 3×1 network for clustering. The displayed numbers correspond to the ARI index.

4 Lab 4

4.1 Stacked autoencoders

		Accuracy on Autoencoder			
Layers	Hidden units	Before tuning	After tuning	Regular net accuracy	**Improvement**
2	[100]	98.38	98.92	97.60	**1.35**%
3	[100 81]	78.82	99.92	96.68	**3.35**%
4	[100 81 64]	38.22	99.78	96.92	**2.97**%
5	[100 81 64 49]	21.50	99.40	97.58	**1.87**%

Table 5: Evaluation of different deep architectures. *Hidden layers* indicates the neurons on each layer (from the first to the last), on both the autoencoders and the regular feed forward net. The final column shows the improvement of the stacked architecture over the feed forward net, in digits classification. For each row, both type of nets share the same structure

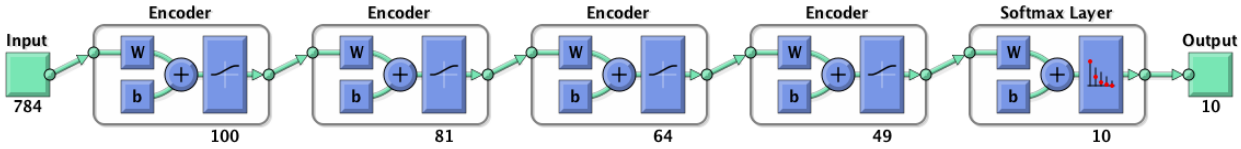


Figure 10: 5 layer architecture of autoencoders, corresponding to last row of Table 5

The goal of this section is understanding the behaviour of autoencoders, regarding their performance compared to regular nets, as well as interpreting what they do.

Table 5 shows that autoencoders outperform regular deep feedforward networks, even when sharing the same structure. Furthermore, they offer a more efficient representation of data, which could eventually used to save storage space. The main drawback is the time that the layerwise greedy training takes, compared to training a simple network using backpropagation. Also, fine tuning makes a big difference in performance, particularly for deeper architectures. It is not good to skip it. Very large architectures are usually trained using GPUs.

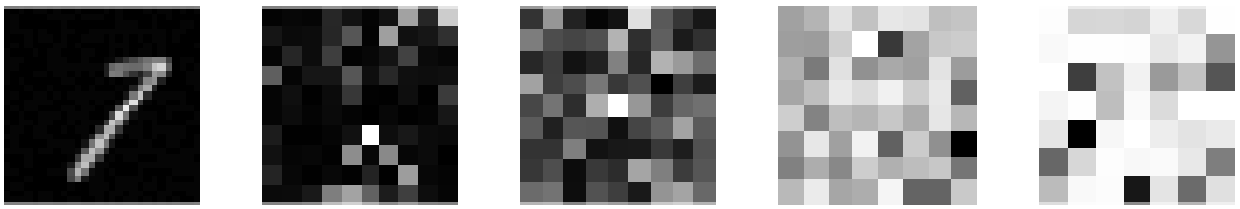
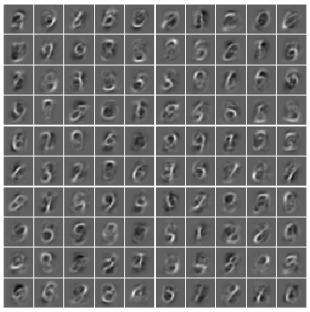


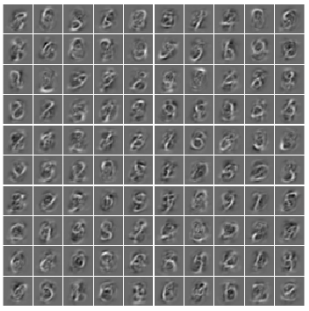
Figure 11: Original digit (left), and the features that each autoencoders extracts from it, using the 5 level deep architecture of Table 5. As depth increases, features become more abstract and hard to understand for humans. Also, their dimensionality is reduced (less pixels on the image). From left to right, images correspond to autoencoders 1 to 4.

Stacked autoencoders extract abstract features from data, in a similar way the brain does. The deeper the architecture, the more abstract features become. Figure 11 showcases this idea: the features, represented as images, that each layer obtains have no meaning for an human. This is actually one of the main problems of deep learning: they often produce *black box* models, and we cannot understand what it actually represents. Therefore, they cannot be used at applications that require transparency, like credit scoring. However, this abstraction is their main advantage: as I showed in the previous paragraph, it beats regular nets with the same structure.

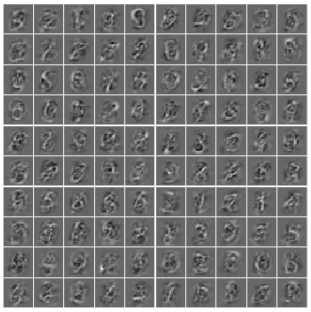
Finally, Figure 12 shows the weights of the first layer after training with different L2 penalties. Higher penalties make the model simpler, so weights are more similar to actual digits. Lower penalties make weights



(a) $L2\ weight = 2 \times 10^{-3}$



(b) $L2\ weight = 1 \times 10^{-3}$



(c) $L2\ weight = 5 \times 10^{-4}$

Figure 12: Weights of the first autoencoder. This plot is very analogous to that of the components in Figure 8: it displays the internal abstract representation the data, but using weights instead of components. Each image is the result of a different L2 regularization weight.

have a higher variance (the model is more complex), therefore they look blurrier on the right image. This is analogous to the ideas I explored in the previous section, particularly on Figure 8: each weight, or component in the case of PCA, model different characteristics of the data (cursiveness, thickness, etc.). This is also one of the reasons for the popularity of deep neural networks: they perform automatic and parametrized feature extraction and selection.

4.2 Convolutional Neural Networks

The `CNNEx.m` script runs a convolutional net on a database with images of laptops, trains and boats, and tries to classify them, as well as show their internal representation. The dimension of the weights in layer 2 is [11 11 3 96], with stride [4 4] and no padding. This is the convolutional filtering layer. The stride is the step size for traversing the input vertically and horizontally. In this case, [4 4] specifies a vertical step size of 4 and a horizontal step size of 4. The first two numbers are the filter size (11x11): width and height to scan the image, 121 pixels in total. The third number (3) is the number of channels. The image is coloured, so they are the RGB channels. The last number, 96, is the number of filters, I.e the number of neurons that corresponds to a region of the output.

On the second question, the fifth layer performs Pooling, using a stride of [2 2] and a filter size of [3 3], with no padding. Starting on the first layer, the input is 227×227. The output after first convolution is:

$$\frac{(Size - Filter + 2 \times Paddding)}{stride} + 1$$

So: $(227-11)/4+1 = 55$, so there are 55×55 neurons. They go then through the pooling layer, obtaining $(55-3)/2+1 = 27$, so layer 6 expects an input of 27×27.

Applying this iteratively returns that on the last layer there are 71×71 feature maps, so 5041 neurons. After the first layer there were $11 \times 11 \times 3 \times 96 = 34848$, which is a big improvement. These are all fed to the final classification layer, a softmax function with 1000 neurons (classes) to output.

```
25x1 Layer array with layers:

   1   'data'     Image Input                   227x227x3 images with 'zerocenter' normalization
   2   'conv1'    Convolution                   96 11x11x3 convolutions with stride [4  4] and padding [0  0  0  0]
   3   'relu1'    ReLU                          ReLU
   4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per element
   5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
   6   'conv2'    Convolution                   256 5x5x48 convolutions with stride [1  1] and padding [2  2  2  2]
   7   'relu2'    ReLU                          ReLU
   8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per element
   9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
  10   'conv3'    Convolution                   384 3x3x256 convolutions with stride [1  1] and padding [1  1  1  1]
  11   'relu3'    ReLU                          ReLU
  12   'conv4'    Convolution                   384 3x3x192 convolutions with stride [1  1] and padding [1  1  1  1]
  13   'relu4'    ReLU                          ReLU
  14   'conv5'    Convolution                   256 3x3x192 convolutions with stride [1  1] and padding [1  1  1  1]
  15   'relu5'    ReLU                          ReLU
  16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
  17   'fc6'      Fully Connected               4096 fully connected layer
  18   'relu6'    ReLU                          ReLU
  19   'drop6'    Dropout                       50% dropout
  20   'fc7'      Fully Connected               4096 fully connected layer
  21   'relu7'    ReLU                          ReLU
  22   'drop7'    Dropout                       50% dropout
  23   'fc8'      Fully Connected               1000 fully connected layer
  24   'prob'     Softmax                       softmax
  25   'output'   Classification Output         crossentropyex with 'tench', 'goldfish', and 998 other classes
```

Figure 13: The architecture from the covnet.

5 Final project 1: regression and classification with MLP

My name is Mauro Paradela del Río, student number r0696150, Master in Artificial Intelligence. My data has been generated using:

$$Tnew = (9T1 + 6T2 + 6T3 + 5T4 + 1T5)/(9 + 6 + 6 + 5 + 1)$$

5.1 Regression

For this exercise I have to estimate a function generated using my student number, using the formula above. The result of this operation is a combination of non linear functions, that is also non linear. The result is plotted in Figure 14.

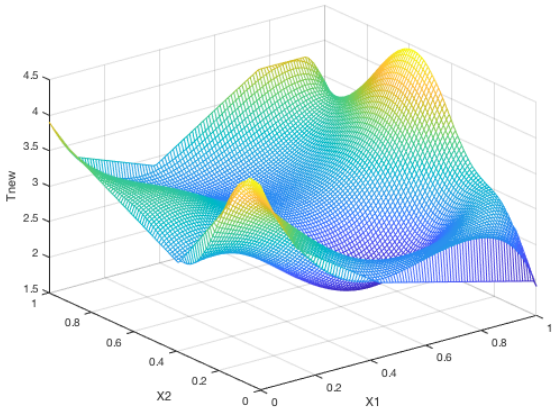


Figure 14: Function produced using my student number

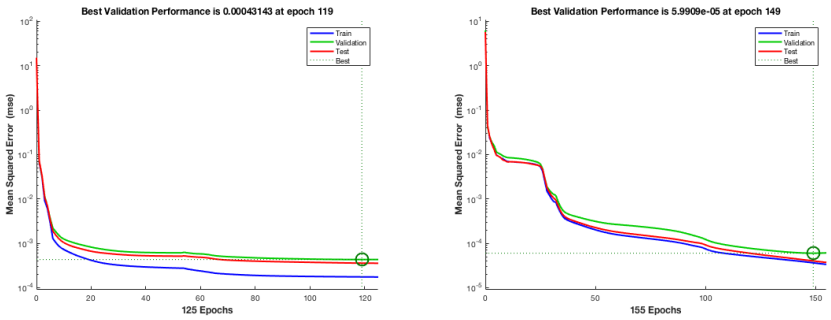
	tansig					logsig				
N	R_t	MSE_t	MSE_{tr}	Epch	T(s)	R_t	MSE_t	MSE_{tr}	Epch	T(s)
25	0.99	9.79×10^{-5}	2.39×10^{-5}	169	11.08	0.99	4.53×10^{-6}	1.46×10^{-6}	200	16.19
50	0.99	1.67×10^{-5}	6.90×10^{-6}	171	10.64	0.99	6.11×10^{-6}	1.50×10^{-6}	200	16.87
100	0.99	1.05×10^{-5}	1.34×10^{-6}	200	20.15	0.99	9.63×10^{-6}	6.17×10^{-6}	160	11.55
150	0.99	2.73×10^{-6}	8.29×10^{-7}	200	16.54	0.99	1.45×10^{-5}	4.80×10^{-6}	172	11.96
200	0.99	5.38×10^{-6}	1.71×10^{-6}	200	12.55	0.99	1.45×10^{-4}	5.89×10^{-6}	167	15.66
250	0.99	1.40×10^{-5}	3.42×10^{-6}	200	12.49	0.99	5.07×10^{-5}	3.24×10^{-5}	165	12.49
300	0.99	1.35×10^{-5}	2.29×10^{-6}	198	12.54	0.99	3.79×10^{-6}	1.14×10^{-6}	200	15.62

Table 6: Metrics on different network configurations, always with one hidden layer and trained using `trainlm`. I compare two different transfer functions on the first layer. All results are the average of 20 runs. N stands for the hidden units,t stands for *test* and *tr* stands for training. It looks like any model is good for this dataset.

Table 6 shows a comparison of different network architectures, focusing particularly on the transfer function and the number of hidden units. For the experiments I have taken 1000 points as validation set, another 1000 for training and 1000 for testing, avoiding overlapping between them. Apparently, a simple network of 25 hidden units with any transfer function produces already a very good result: on the train set, MSE is below 1×10^{-4}, and the correlation with that same set is above 0.99. The only remarkable fact is that `tansig` converges slightly faster than `logsig`, but error seems to be higher. Figure 15 showcases it on the MSE curve evolution: `logisg` overcomes `tansig` at 50 epochs, but takes more iterations to reach its best performance. In contrast, The former is already semi-optimal at 20 epochs, the latter descends steeply at 30.

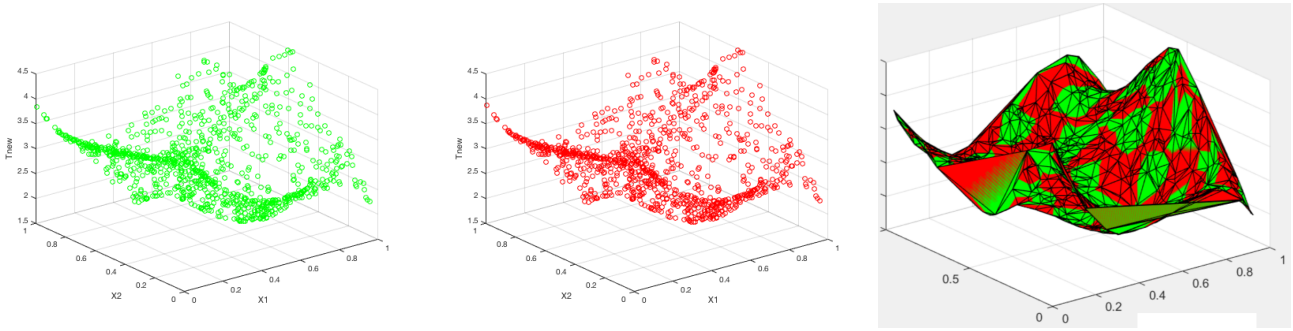
I believe that the non linear structure of the function is well captured by the net. Furthermore, since all values are positive, `logsig` approximates it very well. A way of improving the results would be training

under Bayesian Optimization: since so few neurons are required, it is easy to overparametrize. Bayesian optimization would reduce overfitting in that case.



(a) `tansig` (b) `logsig`

Figure 15: MSE curve on all sets, for different transfer functions on the first layer. There are 25 hidden units, trained using the `trainlm` algorithm. Both behave differently: `tansig` converges faster, but `logsig` achieves better results.



(a) Test set (b) Prediction (c) Both together (interpolated)

Figure 16: Test set (green) and network prediction (red) comparison. 25 hidden units, `logsig` transfer function. MSE on this set is below 1×10^{-5}

5.2 Classification

In this part of the assignment I have to build a feed forward neural network to classify wine into categories. The last number of my student ID is 0, so I have a binary problem with 2198 data points on the negative class (label 6) and 1457 data points on the positive class (label 5), taken from the white wine subset.

For this task I have focused on Matlab's *patternnet*: a feed forward net with a *softmax* function on the output layer. That way the network results ranges between 0 and 1 for each output neuron. When using a *one-hot* encoding for the classes, they can be interpreted as probabilities of belonging to one class or the other. Therefore, in this particular problem all target values are bidimensional vectors, either [0 1] or [1 0].

Choosing the optimal architecture On Table 7 I have tested different training algorithms for pattern estimation. Some of them are new, like Scaled Conjugated Gradient (*trainscg*) and Resilient Backpropagation (*trainrp*). The results show once again that Levenberg-Marquardt delivers the best results. However, this time it is by far the slowest algorithm: it takes up to 50 times more then *trainscg* when using 80 hidden units, for example. Even though it does not calculate the whole Hessian matrix, the lm is too slow when doing pattern classification, even though it was the best performing algorithm for function estimation (see Table 1). The figure also shows that 5 neurons are enough for delivering a decent CCR, although it does not overparametrize very fast: the value keeps constant for the configurations I have tested.

Nhid	**trainscg**			**trainrp**			**traingd**			**trainlm**			**trainbfg**		
	CCR	*Epch*	$T(s)$	*CCR*	*Epch*	$T(s)$	*CCR*	*Epch*	$T(s)$	*CCR*	*Epch*	$T(s)$	*CCR*	*Epch*	$T(s)$
5	0.70	25	0.36	0.70	38	0.27	0.60	431	2.66	0.71	14	0.81	0.72	9	0.54
10	0.70	22	0.41	0.70	48	0.34	0.59	461	3.06	0.72	8	1.03	0.74	14	0.61
20	0.70	23	0.49	0.70	46	0.43	0.59	440	3.87	0.73	10	3.43	0.72	20	1.43
40	0.70	19	0.56	0.71	45	0.68	0.59	478	7.40	0.74	14	16.84	0.71	31	6.89
60	0.70	17	0.76	0.71	41	0.79	0.60	457	7.72	0.73	10	28.42	0.70	32	28.86
80	0.70	18	0.90	0.71	44	1.11	0.60	478	10.10	0.75	10	52.12	0.70	17	39.84

Table 7: Comparing different units on the hidden layer and training methods for the feedforward *patternnet* on the white wine dataset. This time, the most accurate algorithm is not the fastest; quite the opposite. *CCR* stands for Correctly Classfied Data on the validation set, and *Epch* for epoch of convergence in the training phase. Validation and test set are each 15% of the data points.

Dimensionality reduction using linear PCA has two potential benefits: it can save memory and computations by reducing the size of the data, and it can find directions in which classification can deliver better results. According to Figure 17, 3 components are already a very good summarization of the data: they contain most (almost 100%) of the variance, and reconstruction error using 3 components is almost 0. So, the first goal is achieved. However, classification does not improve: as Table 8, no combination of hidden units and transfer functions outperforms the results from Table 7, in this case for Resilient Backpropagation. I have even tried different transfer functions on the first layer, but accuracy is on all of them around 15% lower than it was. I believe that PCA is not finding an optimal direction where classes can be separated well. Figure 18 proofs it: in all the extracted 3 components both classes have a large overlapping.

I would like to point out that I have not changed the transfer function on the last layer (`softmax`) because it is the right one for a classification task. A linear activation function would deliver values over 1 and below 0, and `logsig` is not suitable for multiclass problems.

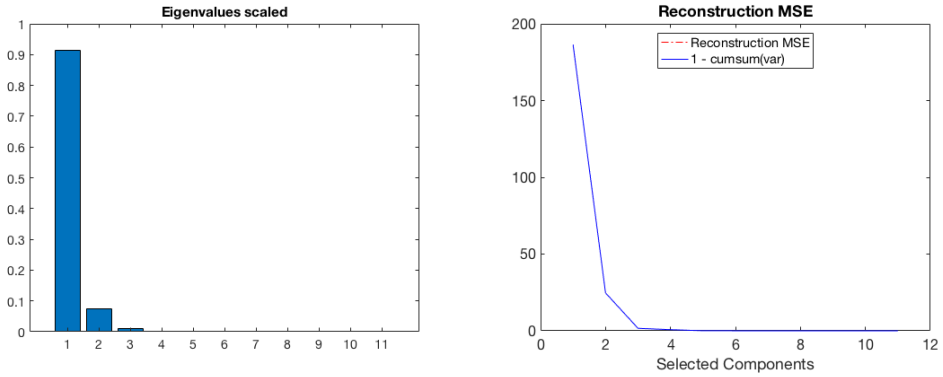


Figure 17: Scaled eigenvalues for the white wine dataset (left) and reconstruction MSE with increasing number of components. Clearly 3 components make a good summarization of the data; they account for more than 95% of the variance

This is not necessary a problem: after all, data is now 3.5 times smaller. Furthermore, when using 6 components (half of the original 11), classification accuracy is at 69%. That means that there are still many features in the data that are not relevant for the model, and can be pruned out. So it is possible to achieve a good compromise between performance and data compression.

Finally, I would like to point out that I have achieved significant improvements when making the net deeper. Using the same configurations as Table 8, and [10 5] hidden units in each layer, I already achieved 68% *CCR* with 3 PC. This improvement happened on all the settings, but only with two layers. With more layers my model performed worse than using only one, not even delivering the majority class.

Nhid	`logsig`		`tansig`		`purelin`		`softmax`	
	CCR	$T(s)$	CCR	$T(s)$	CCR	$T(s)$	CCR	$T(s)$
5	0.62	0.27	0.61	0.19	0.60	0.15	0.62	0.23
10	0.61	0.23	0.62	0.22	0.61	0.12	0.61	0.31
20	0.62	0.37	0.62	0.27	0.60	0.15	0.62	0.34
40	0.62	0.39	0.61	0.55	0.61	0.27	0.62	0.82
60	0.62	0.52	0.62	0.61	0.61	0.32	0.61	1.39
80	0.62	0.63	0.62	0.81	0.61	0.36	0.62	2.31

Table 8: Classification performance on the reduced white wine dataset (3 PC), using a *patternnet* with one hidden layer and trained with Resilient Backpropagation.

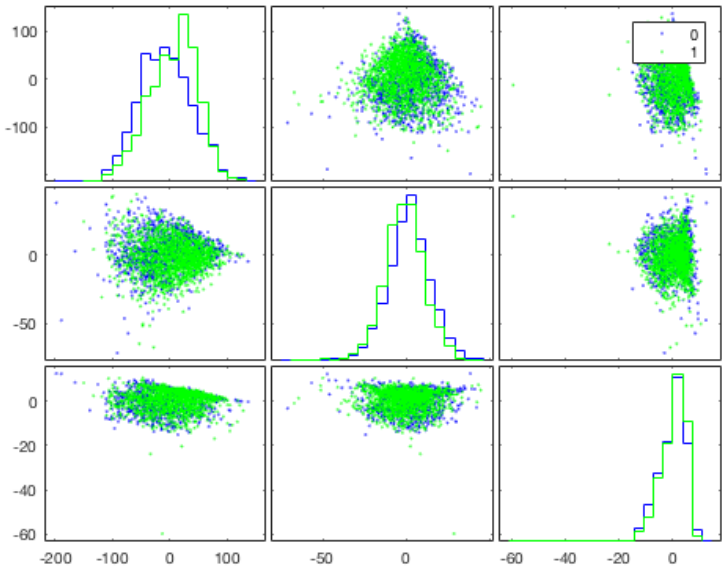


Figure 18: Gmatrix plot of the reduced white wine dataset. No component separates well the classes. Class 0 corresponds to negative class, class 1 to the positive 1.

6 Final Project 2: Hopfield Networks for character recognition

The goal of this section is investigating the capabilities of Hopfield networks for character recognition. For this purpose, I have created a collection of characters made by the 26 capital letters of the alphabet, plus the 10 than make up my name and surnames. I am Mauro Paradela del Río, that means: m, a, u, r, o, p, d, e, l, i. This letters are displayed in Figure 19. They are stored as figures of size 5×7.

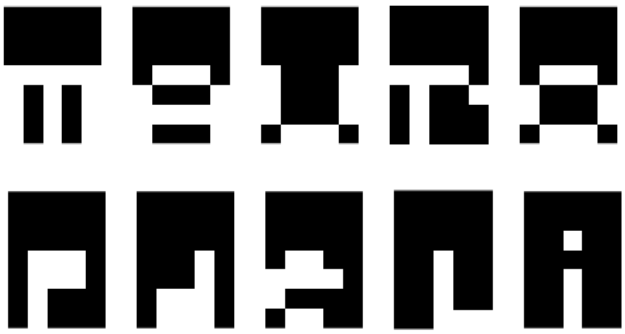


Figure 19: First 10 stored characters, corresponding to the 10 different letters in both my name and last name.

Task 1 I performed 50 different experiments flipping random pixels on the characters, and I iterated the network 100 times on each. I never found spurious states: I could always recall an attractor of the net. However, some of my letters were very similar, particularly the u and the o: depending on the noise, one of those two letters could end up resembling the other. An example of this is shown in Figure 20. This happens often because of the low dimensionality of the collection: with just 35 pixels, it is very easy that the characters correlate. Flipping 3 means modifying almost 10% of the data.

This does not mean that there are not spurious patterns, I probably was just (un)lucky and did not find them. The critical load of a Hopfield network (storable patterns P) is given by:

$$P < \frac{N}{4ln(N)}$$

Being N the number of neurons and P the storable patterns. For $N = 35$, P is smaller than 3.

Task 2 Figure 21 shows the count of wrong pixels retrieved over different numbers of stored patterns. I started with 1 and went up to 25 patterns, and I let the net converge to a stable state (at least 100 iterations). From this and other runs it seems that the net starts to make mistakes between 8 and 10 patterns. Therefore, I estimate the empirical critical load capacity of the net as 8. This is higher than the theoretical threshold from before: probably at that point attractors are not so similar, and the network finds it easy to differentiate between them.

Task 3 It is not possible to store 25 characters using just 35 neurons, even without inversions. Using a regular Hopfield Network, it would be necessary to increase the number of neurons to at least 290.

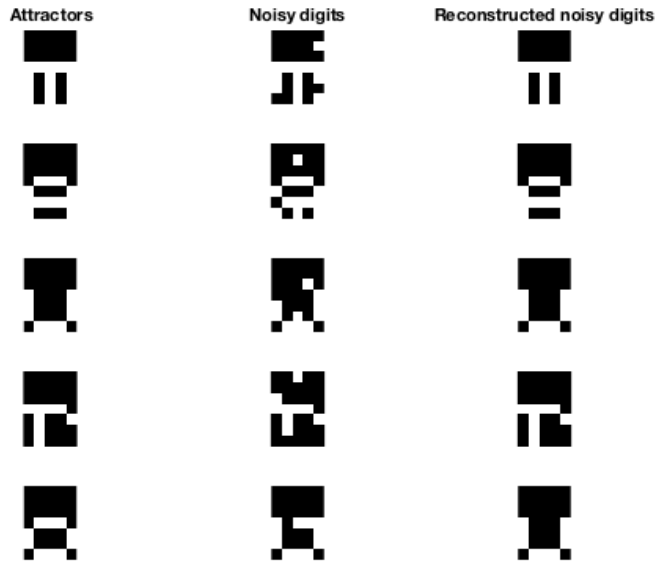


Figure 20: First 5 letters, stored as attractors, and their corrupted version. On the right column, the retrieved character. Note that the last character (the o) is wrong.

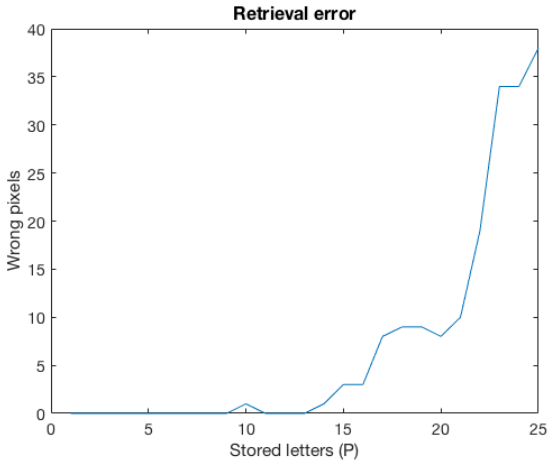


Figure 21: Error on the retrieved pixels over all the stored patterns