Report

Andrés Reverón Molina – r0767329

Artificial Neural Networks and Deep Learning

# Supervised learning and generalization

## Perceptron

As we know, due to its simple architecture consisting of a single neuron, a perceptron is only able to learn linearly separable tasks, that is, groups of points that can be separated by and hyperplane in the input space, this is called decision boundary. This makes it a useful tool for basic classification tasks, such as spam detection. On the other hand, a perceptron cannot learn non-linear functions, such as the XOR function.

For this kind of tasks, the perceptron is normally able to achieve a high accuracy score, and it is not computationally expensive at all compared to other network structures. In fact, it is the simplest network consisting of only one layer.

## Backpropagation in feedforward multi-layer networks

The limitation of linearity comes to an end when several layers are introduced. A simple multi-layer network consisting of one hidden layer and one output layers is perfectly capable of being an universal approximator, provided that a sufficient number of hidden units is taken(neurons). Normally, a network with two hidden layers is chosen, paired with an activation function in the output layer, such as tanh or a Radial Basis Function(Gaussian).

A nice advantage of multi-layer networks over polynomial expansion is that they are able to cope better with the curse of dimensionality. This is because the approximation error of the network is virtually independent of the dimionality of the input space, as opposed to polynomial expansion.

The first algorithm that was ever invented for multi-layer perceptrons is called the backpropagation algorithm. It caused a revolution in the area of neural network research. The network optimizes a so-called objective or energy function, the Mean Squared Error(MSE) on the training set(as it is a supervised learning technique). Explained simply, backpropagation is a way of iteratively improving the weights of the network by trying to minimize the value of the error obtained in the previous iteration(epoch). By doing this, the model becomes more reliable, thus improving its ability to generalize.

The algorithm calculates the delta of each unit to see which unit is responsible for most of the error and fine-tune its weight. This is done by, in a sense, sending information back from the output layer, thus effectively backpropagating information.

The backpropagation algorithm is also called generalized delta rule and it is an straightforward way of obtaining an analytic expression for the gradient of the cost function, ever for large networks with a big number of layers.

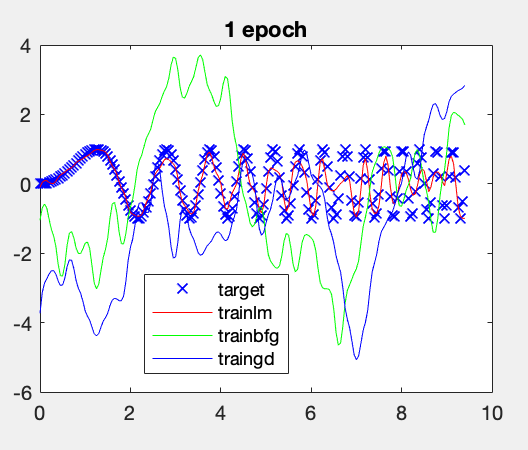
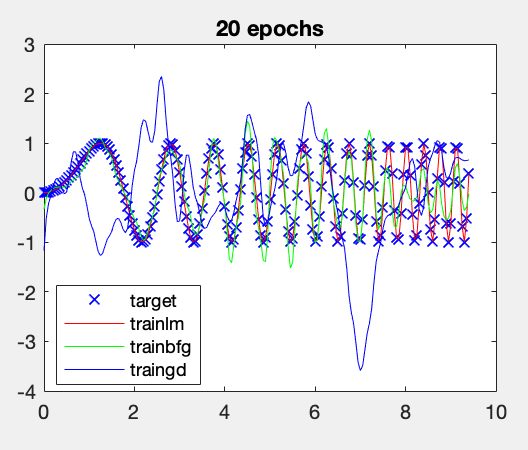
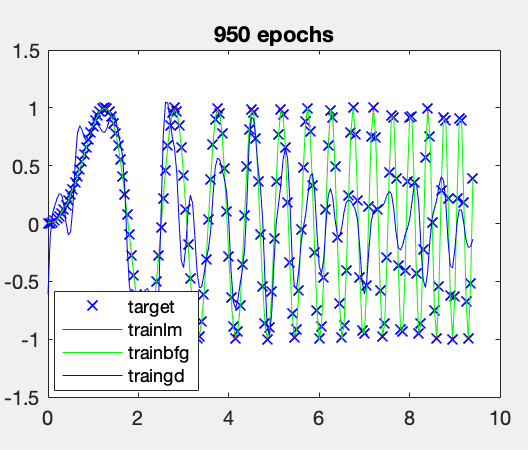
When learning a model, it is important to note that the objective is not to memorize the training data and being able to predict it. The real goal is to approximate the function that is generating this data, in order to be able to generalize this knowledge on new unseen data(the test set). A good model, which is able to generalize, must find a balance in the bias-variance tradeoff. This is basically avoiding overfitting the model with a specific dataset and rendering it unable to generalize. One way of achieveing good generalization without overfitting is via cross-validation.

This is also the goal of the complexity criteria, which states that the complexity f the model should be as low as possible. It is basically the same principle as in the Occam’s razor. Other methods include pruning and ensemble learning.

### Comparison of various algorithms

There exist multiple methods to learn a feedforward multi-layer network with backpropagation. Three of the most popular and discussed methods are the Levenberg-Marquardt(LM), Quasi-Newton(QN) and Gradient Descent(GD) methods, so for the first exercise, we will compare the performance of these three learning methods on a dataset that constists of points generated by a sine function. This task should not be any problem to a neural network of these characteristics, as they are universal approximators, and thus are capable of approximating non-linear functions, as stated in Hornik’s theorem. The three selected methods were trained on the same dataset, with a varying number of epochs(1, 20 and 950 respectively) to see their impact on each model’s accuracy. The figures shown were plotted using 50 neurons in the hidden layer.

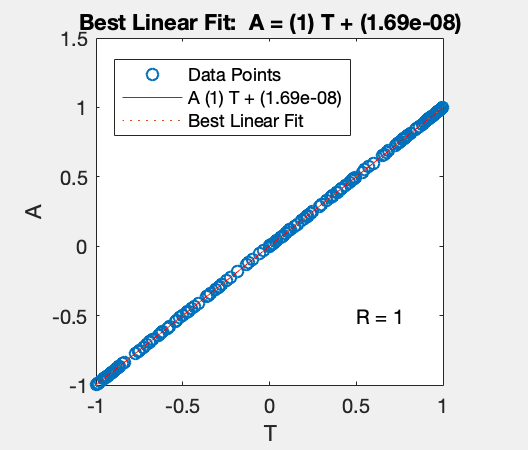
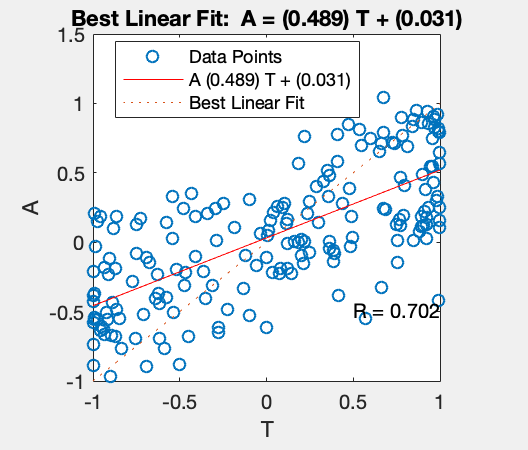
It is important to note that many other learning methods exists, such as Resilient Backpropagation or One Step Secant, but they have been left out of this analysis for the sake of clarity.

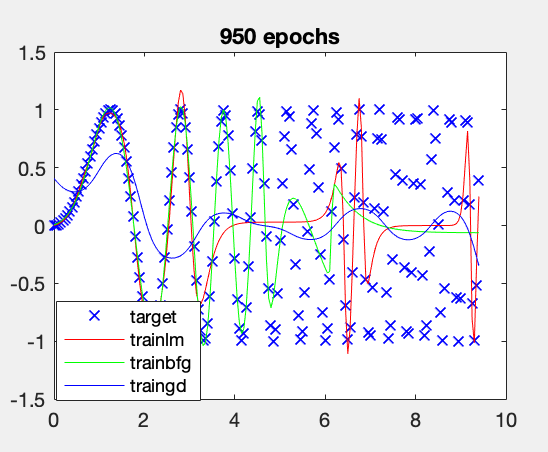
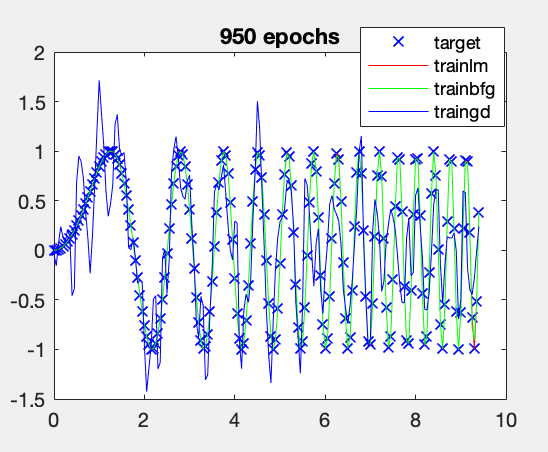
As we can see in the figure, LM achieves by far the best approximation, and it does so quickly, at the cost of more memory usage. Under the same conditions(initial weights, bias, number of epochs and neurons), a pure GD approach gives really disappointing results. This is expected as the LM method if working towards finding the best error reduction, not just the biggest one at a given point.

Given some more epochs, the QN approach also achieves good results, but, again, we can observe that, no matter the number of iterations over the training data, the GD algorithm produces subpar results and it is not able to approximate the input function.

Overall, out of these 3 methods, LM achieves the best results, and also has the best training time. These results are confirmed by performing a regression analysis of the difference between targets and outputs. For example, here is the comparison of the linear fit achieved by LM and GD with 950 epochs:

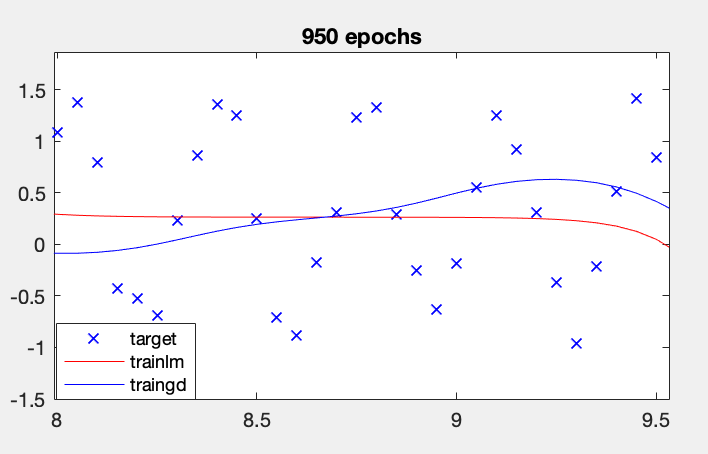
 

The effect of the number of neurons in the hidden layer was also studied. Lowering the number of neurons to 10 translates into significantly worse predictions, while increasing the number of neurons up to 100 exponentially increases the computation cost without affecting the quality of the predictions in a significant way. Although they improve slightly in the case of GD. A good balance between prediction accuracy and computational cost lies around 35 neurons.

### Learning from noisy data

Although these methods work fairly well with the task of approximating a sine function, the results vary tremendously when noise is introduced in the data, and the number of training points in increased slightly. In the following figure, the number of training points has been increase from 3pi to 10 pi, and a random noise factor has been included.

****

We have zoomed to a random location of the plot in order to be able to appreciate the target function points. The difference compared to the previous plots in highly noticeable. Both algorithms are unable to learn the underlying function from the data, thus showing that these methods are not really robust against noise. This is a clear case of overfitting: the models are not able to ignore the noise and thus their predictions do not approximate the function generating the noise.

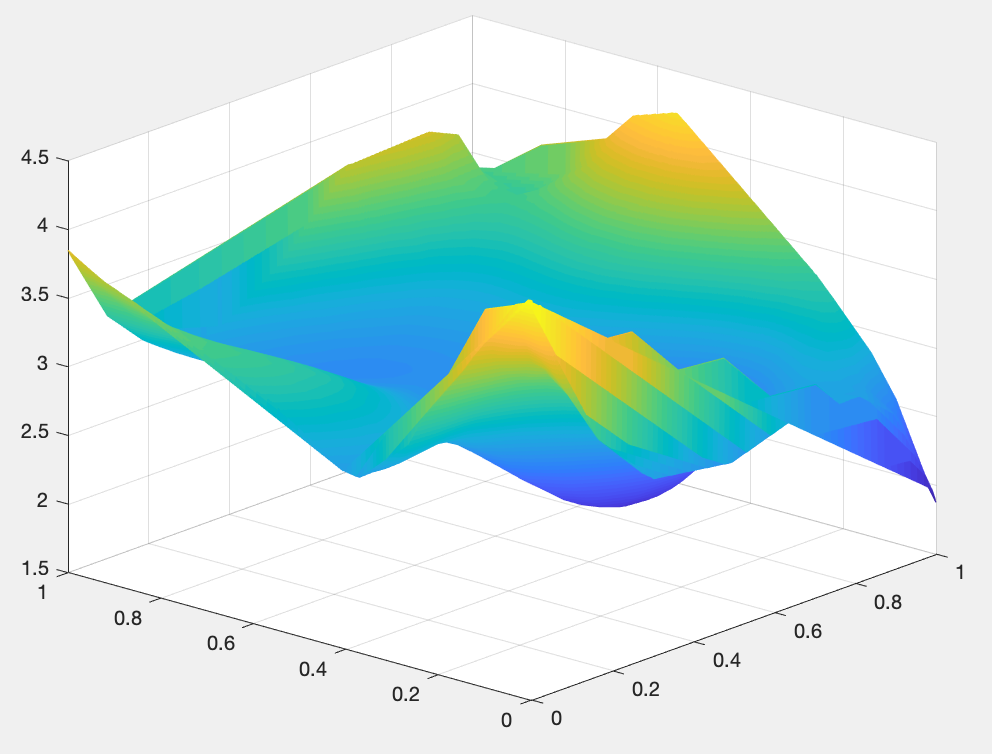
### Approximating an unknown function

Now, we will try to learn an unknown function from data. For this task, a custom dataset was built based on my student number. From an exising dataset T, the custom dataset Tnew was built such as:

Tnew = (d1T1 + d2T2 + d3T3 + d4T4 + d5T5) / (d1 + d2 + d3 + d4 + d5)

Where d1, d2, d3, d4 and d5 are the largest 5 digits from the student number in descending order.

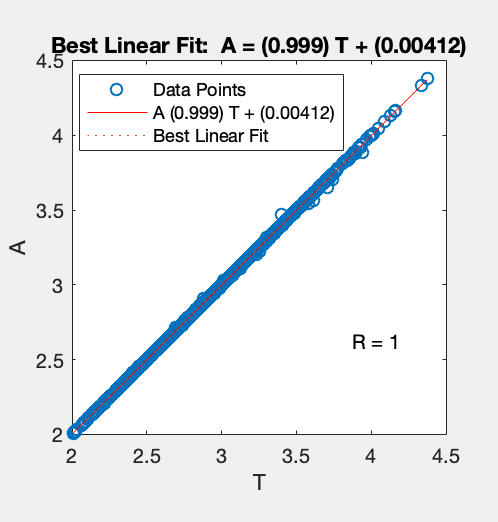
The dataset consists of 3 independent samples of 1000 points each, built with the *datasample* function. The resulting training dataset can be plotted using the mesh function and can be seen in this figure:



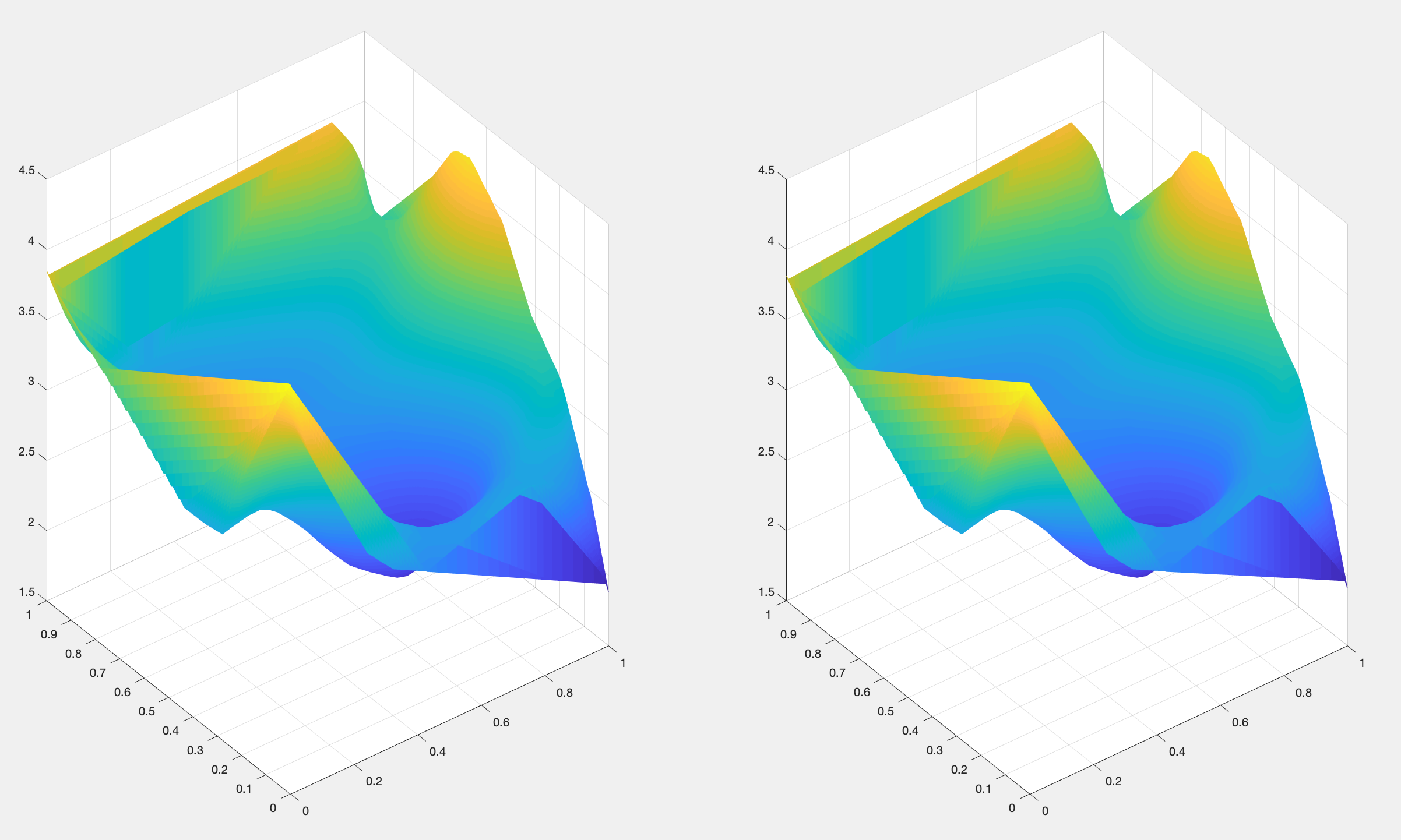
Once the dataset is prepared, we need to find the best possible neural network for the task at hand. In order to find the optimal combination, an algorithm was designed. This algorithm iterates over 6 different learning functions and different number of neurons for the network, and outputs the combination that gives the best R score with the lowest number of neurons possible. We noticed that increasing the number of hidden layers to anything more than 2 hidden layers had a big impact on computational complexity, without improving accuracy significantly, so all tests were performed with 1 and 2 hidden layers.

As these algorithms are non-deterministic, the test was run multiple times and the resulting choice was done via majority vote. As expected based on the previous exercise, the best algorithm for the task was Levenberg-Marquardt, which achieved an R-score of 0.999 with just 15 neurons in two hidden layers.

On the test dataset, the network scores fairly well, which indicates that it is able to generalize the learned concept to new datapoints correctly. Calculating the mean squared error of the predictions a value of almost 0.



Here we can see the actual test points vs the prediction of the model.

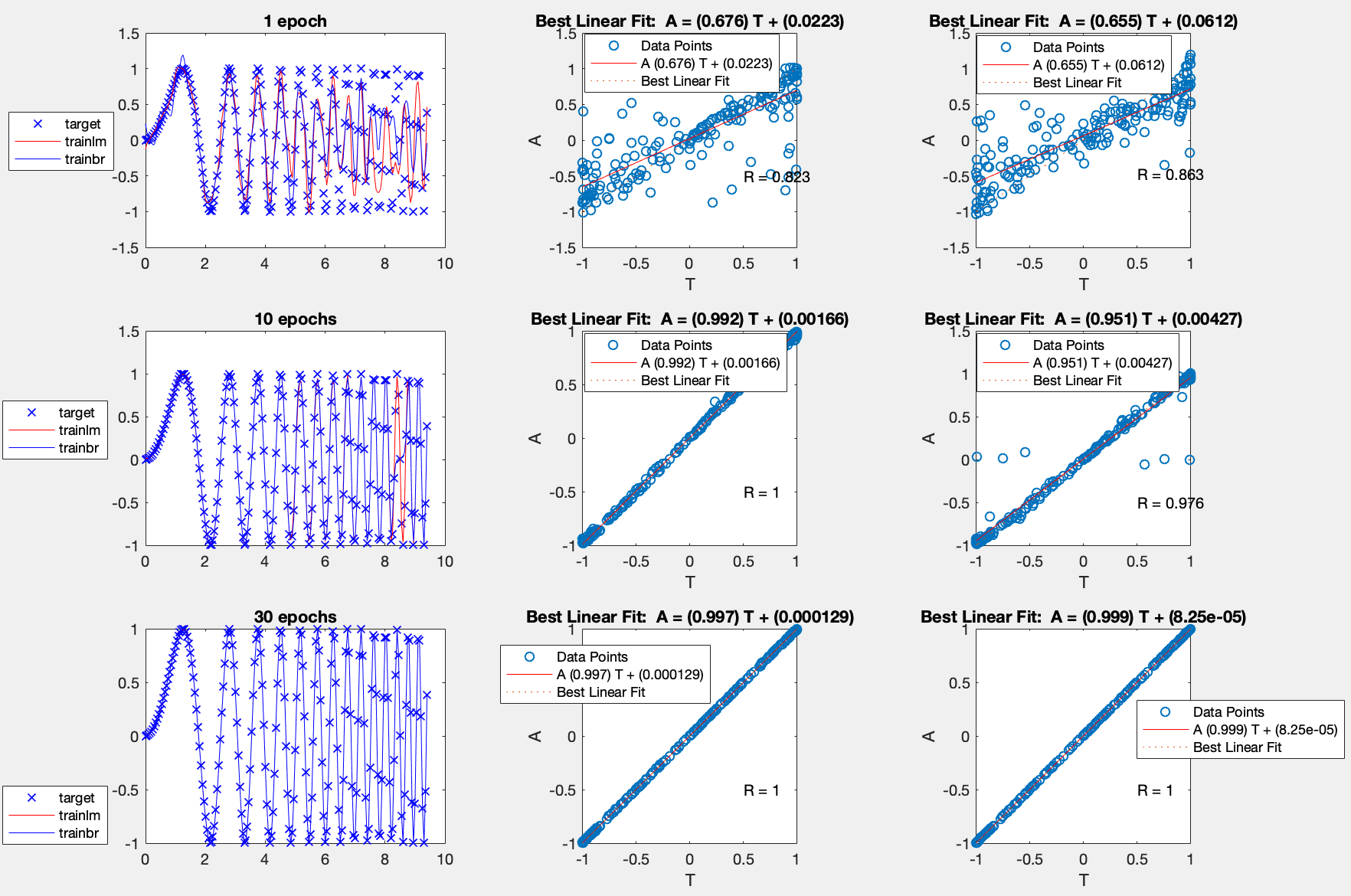


As it can be seen, the figures are almost identical.

### Bayesian Learning Regularization

For the last exercise, we will repeat the analysis performed in the first part of the practical session, but now the Bayesian Learning Regularization(*trainbr*) algorithm will we studied. This algorithm updates the weight and bias values according to the previously studied Levenberg-Marquardt learning algorithm. The main difference is that *trainbr* minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes well. The process is called Bayesian regularization.

Given this definition, one would expect this algorithm to perform better on test data, as it has better generalization properties. Surprisingly, that is not the case, and *trainlm* performed better in every test. On this figure we can see how *trainlm* is consistently better, no matter the number of epochs.



# Recurrent neural networks

## Hopfield networks

A Hopfield recurrent network has one layer with *symmetric saturating linear transfer function(satlins)* and all their neurons are fully interconnected. As a recurrent network, it is part of the non-linear family of learning algorithms. It learns in a synchronous way: an output at time *t* is an input at time *t+1*. A stable Hopfield network has one or more attractors, which are configurations of the neuron values which are not changed by an update of the network. An attractor can also be described as an energy minimum point on the network state. New data points fed to the network will converge to one of these network attractors.

This kind of network is used as associative memory, as it associates new data points to its attractors(points already learnt). This kind of network can learn classification tasks such as digit recognition. The first part of this session consisted of several experiments based on this type of network.

### Simple network

A simple Hofield network with initial attractors [1 1; -1 -1; 1 -1]T was created. Then, several random inputs were fed into the network and we simulated the network with a multiple step iteration consisting of 30 steps.

On average, the network was able to find the corresponding attractors after 16 iterations. The number of total attractors was bigger than the number of attractors used to create the network. The following table show the inputs and their corresponding attractors. As it can be seen, there are a total of 4 attractors, compared to the 3 original ones. The extra attractor is [-1 1].

|  |  |
| --- | --- |
| Input | Attractors |
| [0.4 0.7; -0.6 0.9; -1 0.1] | [1 1, -1 1, -1 1] |
| [0.3 0.6;-0.1 0.8;-1 0.5] | [1 1, -1 1, -1 1] |
| [0.5 -0.6; -0.3 0.1; 1 0.6] | [1 -1, -1 1, 1 1] |
| [-0.3 0.6; 0.1 0.9; 1 1] | [-1 1, 1 1, 1 1] |
| [0.6 0.6; -0.2 0.2; -0.7 0.2] | [1 1, -1 1, -1 1] |

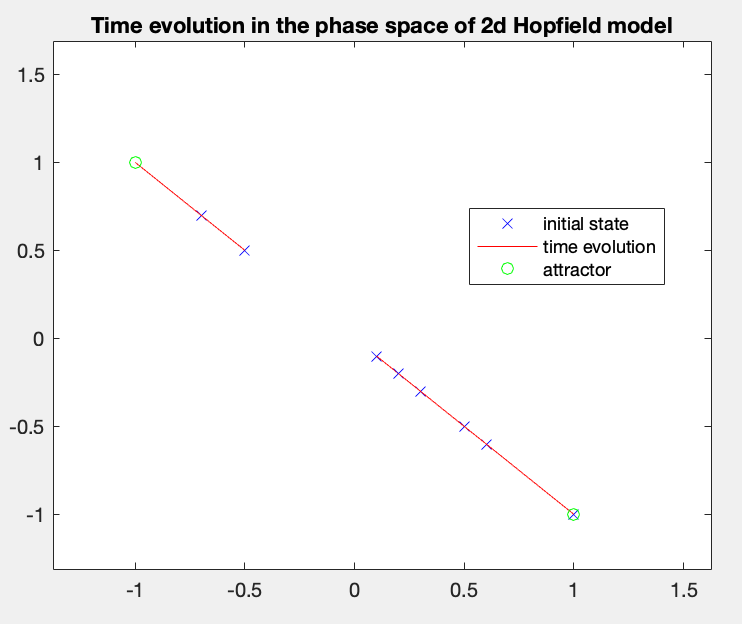
The increased number of attractors can be explained because of spurious patters. This happens when the network converges to local minimum energy points which are not part of the originally defined attractors.

### High symmetry points

Next, we use a Hopfield network initialized with the same three attractors as before, but we fed points with high symmetry into the network. The following matrix was used as input:

[1 -1; 0.5 -0.5; 0.3 -0.3; 0.6 -0.6; -0.7 0.7; -0.5 0.5; 0.1 -0.1; 0.2 -0.2]T

Then, the network was simulated for 50 timesteps, and the evolution was plotted. We can see how the points are iteratively being drawn to their final destination: 2 attractor points. These two attractors are [-1 1; 1 -1]. As it can be observed only one of these attractors corresponds to the initial attractors, and the network has detected 2 attractors instead of 3. These is due to the high symmetry of the input points.



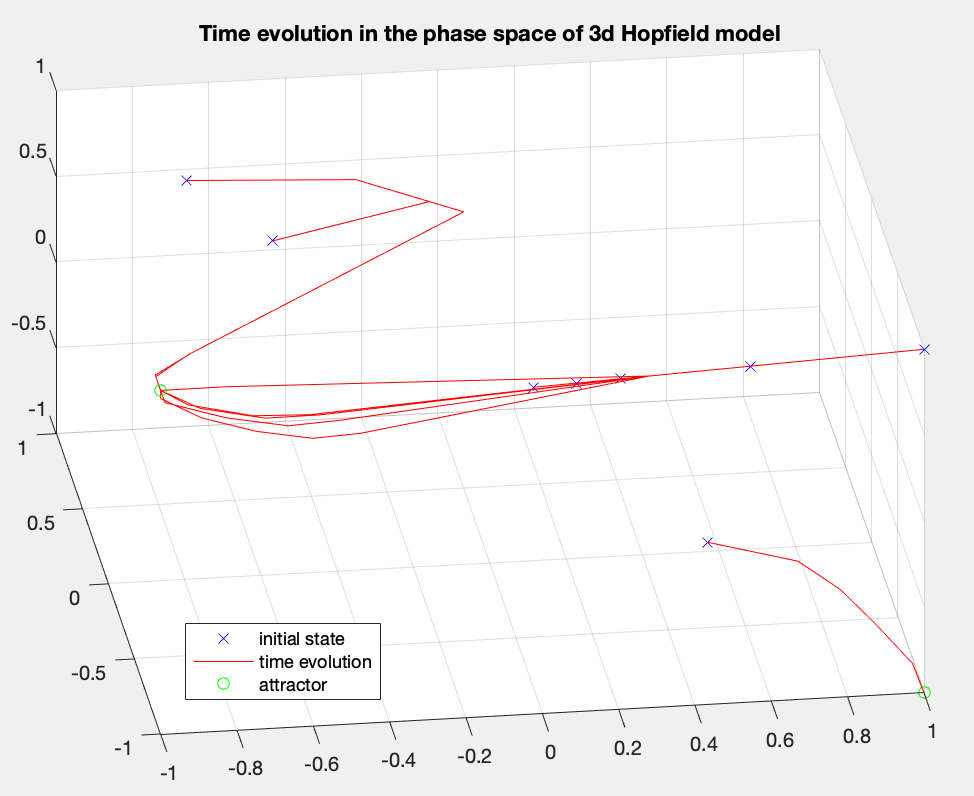
### Three-neuron Hopfield network

Now, we will repeat the process done in point 2.1.2, but the network architecture will contain 3 neurons instead of just 2. With the increasing the number of neurons, an increase of the required computational complexity can be expected. Nonetheless, some improvements are to be expected as well.

The same points were fed into the network, which this time was initialized with the following matrix:

[1 1 1; -1 -1 1; 1 -1 -1]T

Again, we simulated the network for several timesteps and plotted the results. It is important to note that the number of iterations needed to converge to the final attractors is much higher in the 3D space than in the 2D space (250 vs 15). This is expected due to the increase in complexity of the network.

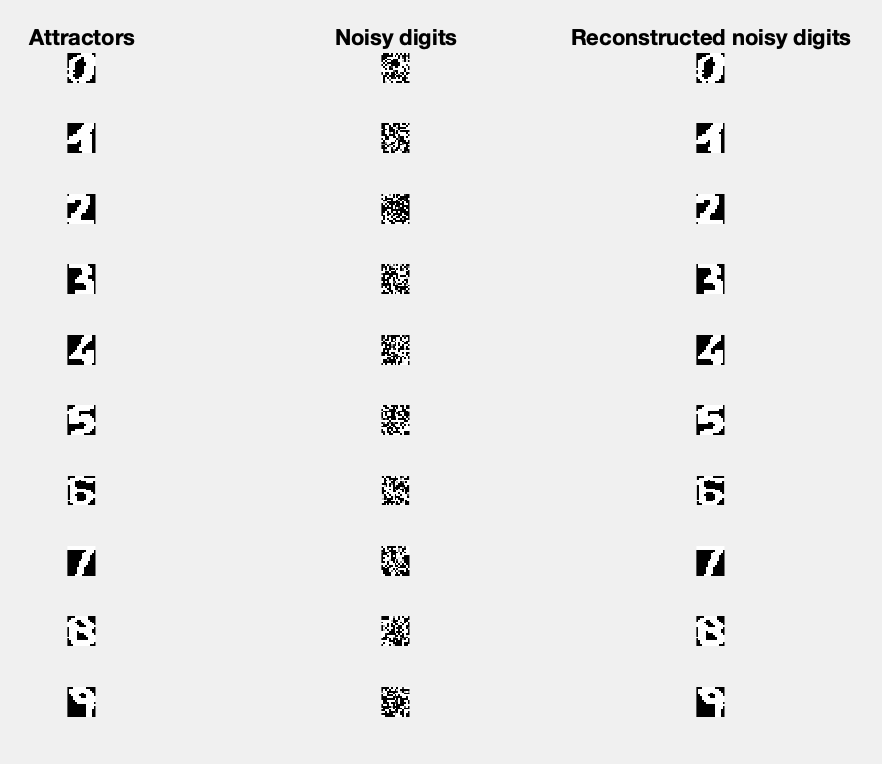


### Handwritten digit classification

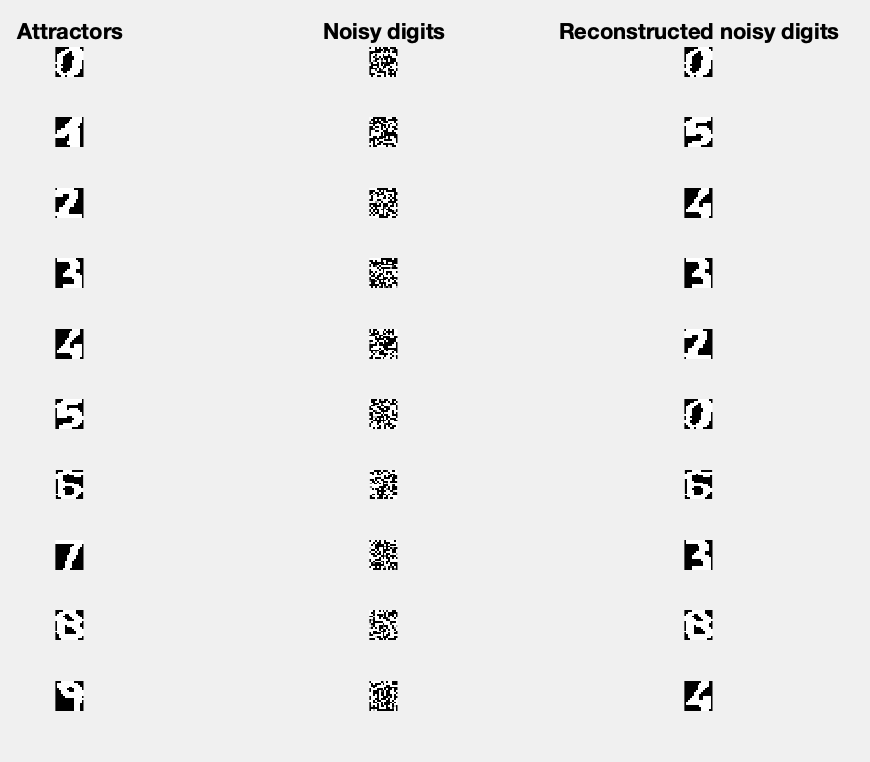
The final exercise about Hofield network is related to handwritten digit recognition. For this task, the function *hopdigit* was used. This function creates a Hopfield network whose attractors are the handwritten digits from 0 to 9.

As not all handwritten digits are exactly identical(they depend on the person writing), some noise is added to the input of the network in order to test its robustness. The level of noise is variable between 0 and 10.

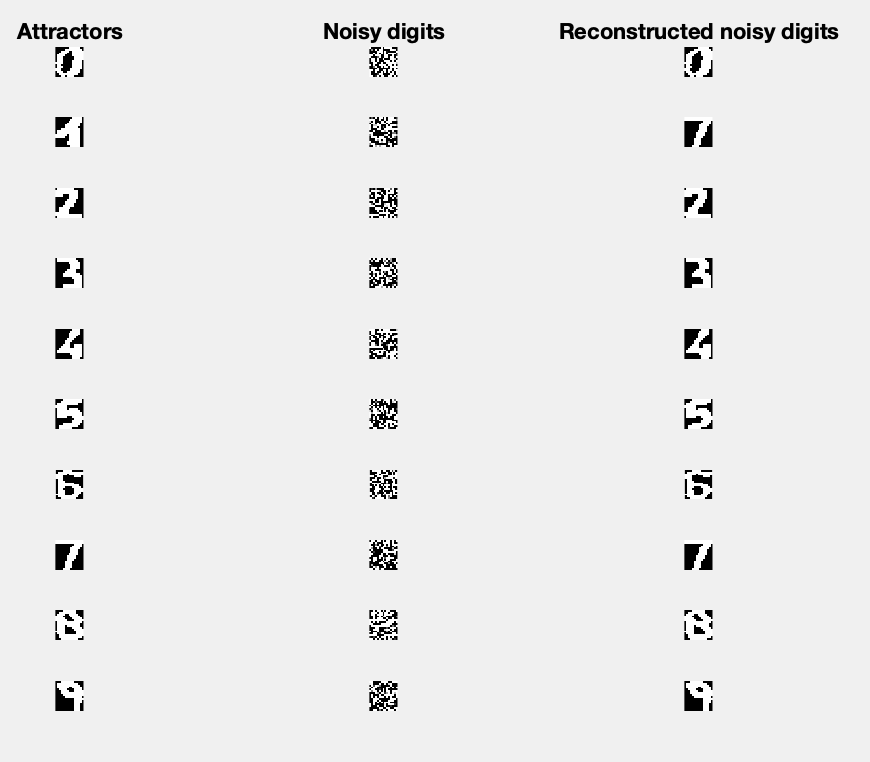
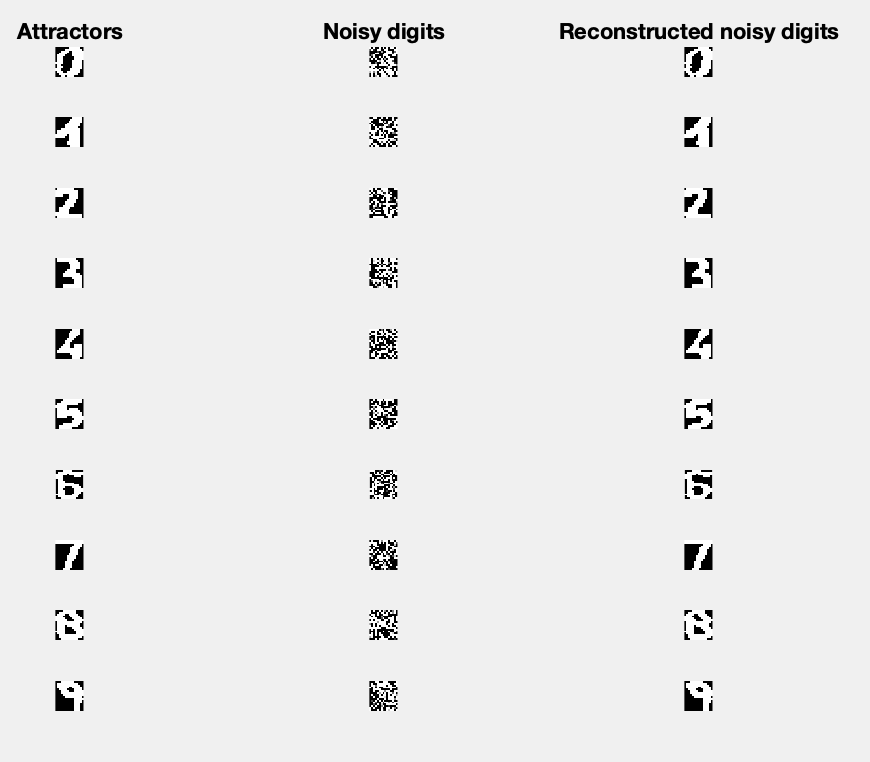
The network is generally very capable of identifying the noisy input. The following figure shows the predictions given a noise level of 5 and 1000 epochs.



However, if the noise level is increased to the maximum value, the model is not able to predict correctly all the values. It starts confusing the numbers 3, 5, 8 and 9. With this level of noise, the model fails to predict the correct values even with a huge number of epochs(9000).



Finally, with a moderate noise(5), increasing the number of epochs does help improving the predictions. Here we can see the comparison between 100 and 1000 epochs:

## Long short-term memory networks

A time series is a sequence of observations ordered in time. In order to predict new points of this type of data, some historical data is fed into the model to train it, for example, a linear auto-regressive(AR) model.

With time-series model, a prediction for a certain time *t* is given by the weighted sum of the data values up to a certain lag time called *p*.

We will first implement a MLP and then compare it with a LSTM network.

### MLP for time-series prediction

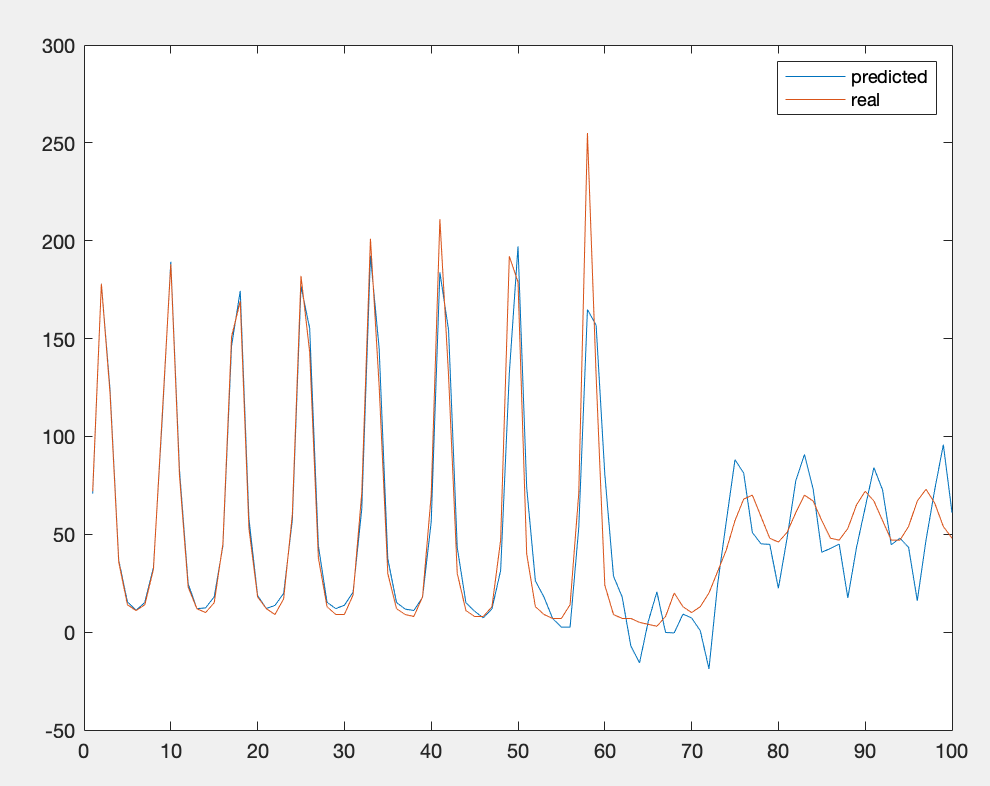
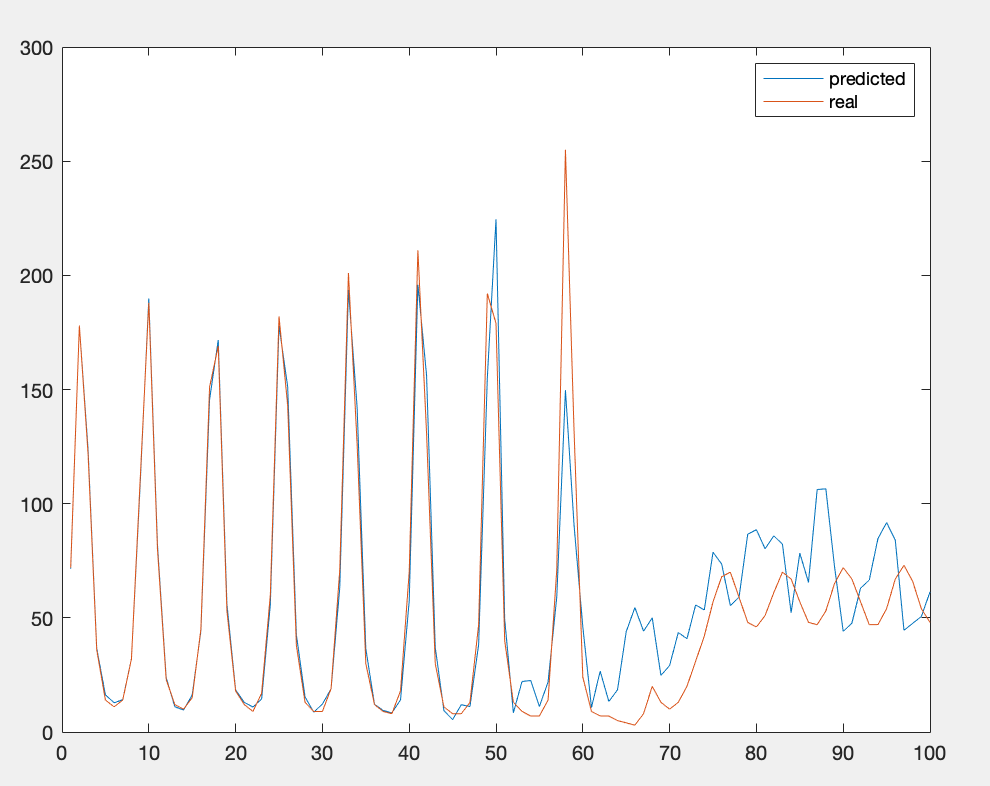
The Santa Fe dataset has been used to train a MLP with one hidden layer in order to predict the next 100 points of the time-series. Several combinations of hidden units, epochs, lag and learning algorithms were tested. In addition, the tests were repeated 10 times and averaged in order to get a representative value.

We found that the models present a really high variability in the results. Even with the same training parameters, a model could give pretty good results one time and pretty bad results if re-trained. In general, the predictions are not optimal, with RMSE values in the range of [20, 60].

The parameter that affected accuracy in the most significant way was the lag value. Anything below 100 gave relatively bad results.

We also compared the trainlm and trainbr learning algorithms, and found that trainbr performs a little bit better, although takes much longer to train. This is expected, as trainbr generalizes better.

In the end, the best combination of training parameters was 40 hidden units, 1000 epochs and a lag value of 100. In the following figure we can see the best predictions obtained with these parameters and the trainlm(left) or trainbr(right) algorithms.



### LSTM networks

LSTM networs are a special kind of network, which is capable of inducting long-term dependencies from time-series data. In the case of the Santa Fe dataset, this feature is really convenient, as the data has some specific points in time with a high variabily with respect to the rest of the points. Thus, knowing the impact of these specific points is of vital importance in order to make accurate predictions.

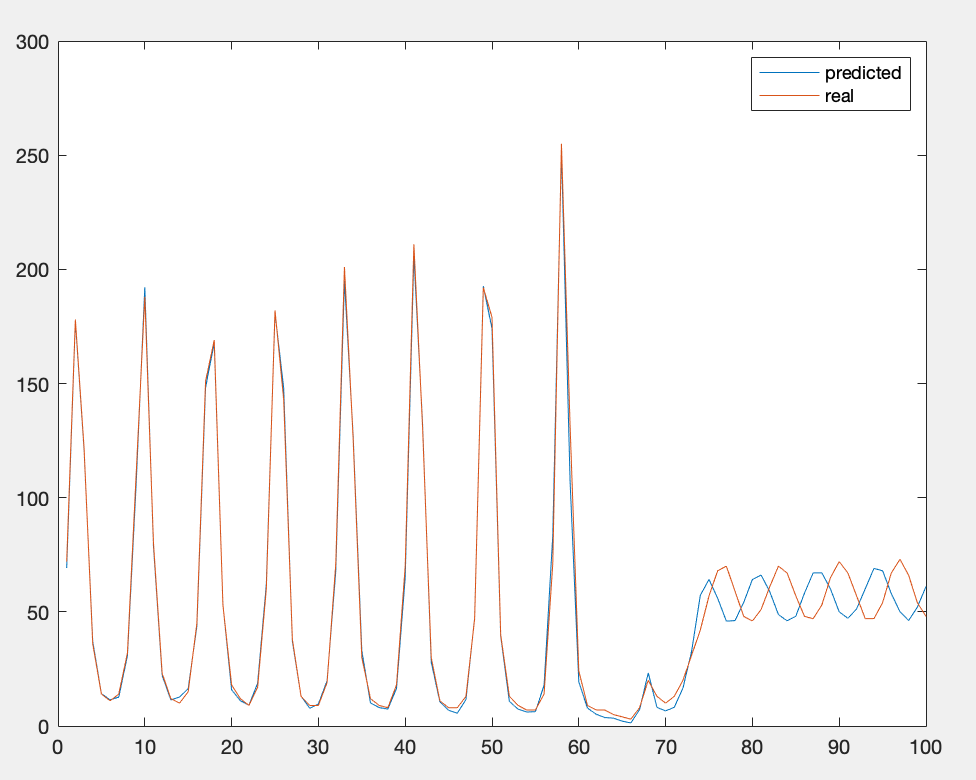
The results achieved by the LSTM network were generally better than those of the MLP, with lower RMSE values. Nevertheless, we found again a high variability in the results; training the network several times with the same parameters gave very different results each time. Therefore, the tests were ran a number of times, and the an average of the results was recorded.

The best found parameter combination is:

* 350 epochs
* 50 lag
* 200 hidden units
* 100 LearnRateDropPeriod

The effect of lag is not as straightforward as with the MLP model. In this case the best RMSE value was obtained with a lag of 60 units. The correlation between RMSE and lag is not linear as it was before. This can be seen in the following figure, which compares RMSE vs lag:

Therefore, the best predictions were obtained with a lag of 60 units. The RMSE value was 8.69 and the results are shown in the following figure:



Seeing the results, it can be said that LSTM networks adapt better than MLP to pattern changes, which makes them a more robust solution for time-series prediction.

# Deep feature learning

## Principal Component Analysis(PCA)

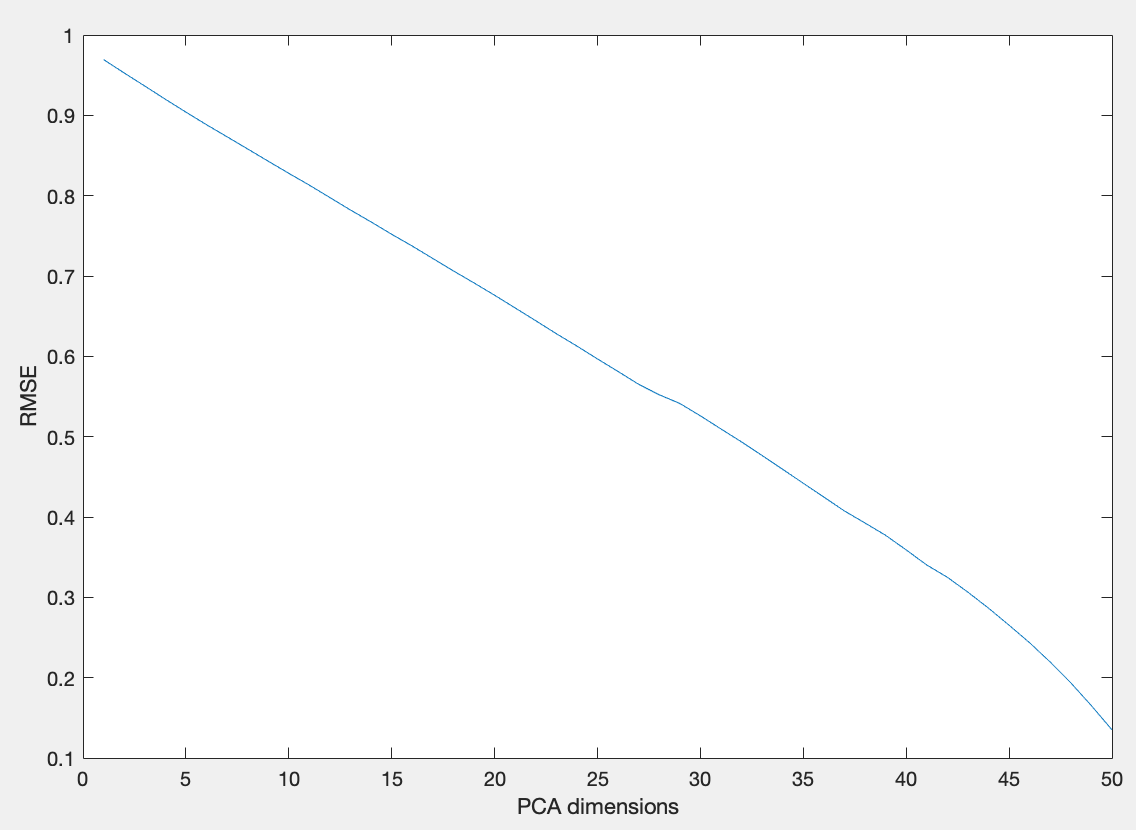
In the first part of this exercise session, we will explore the concept and applications of PCA. Roughly, this technique reduces data dimensionality to a point where data is still meaningful enough to learn patterns from it, but the learning is done in a much faster way thanks to the dimensionality reduction. It is important to remember that *the curse of dimensionality* is one of the most infamous hurdles in the Machine Learning world, and it affects a great amount of leaning models. Therefore, PCA can really come in handy to alleviate this dimensionality problem.

In order to do this, PCA uses the eigenvectors of the covariance matrix to map an input vector of a *p* dimensional space to a lower-dimensional vector. The main idea is that this reduced data can be later reconstructed into the original dimensions, where the reconstructed data should resemble the original data in a significant enough way. PCA can be then seen as a compression algorithm, although information is inherently lost.

### Redundancy and random data

The PCA algorithm was implemented in MATLAB. The algorithm zero-means the data and calculates the covariance matrix. Then, it extracts the eigenvalues and eigenvector from this matrix and the desired dimensionality for the output data. Finally, it reduces and reconstructs the dataset.

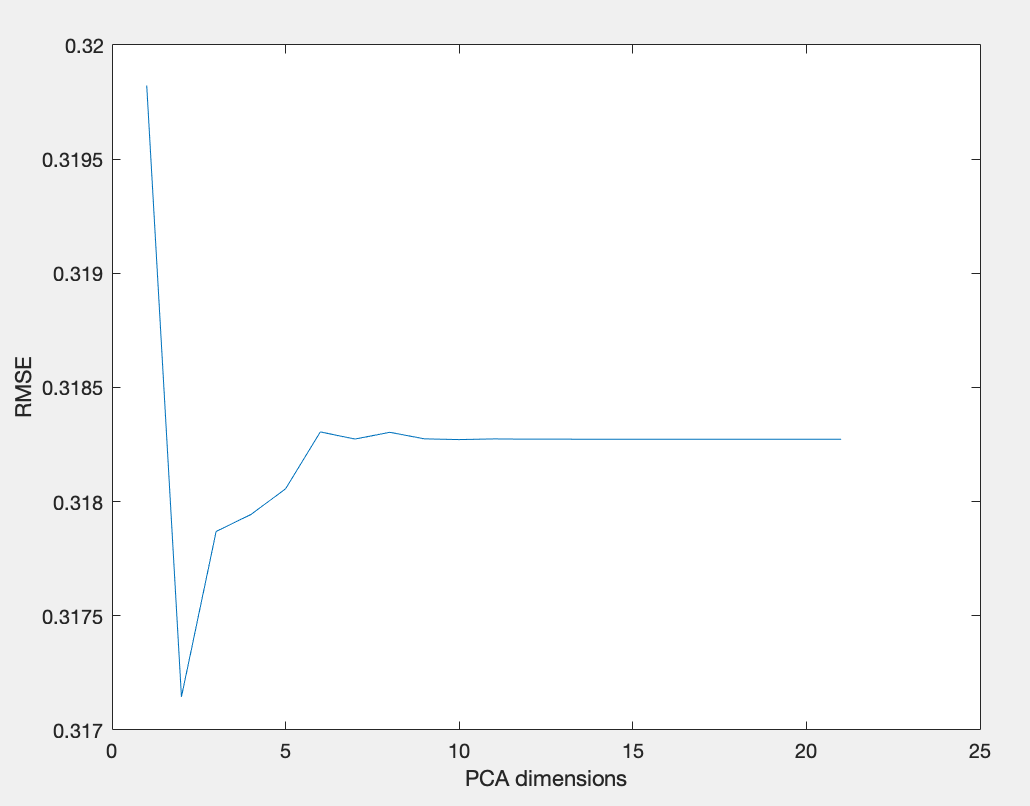
The algorithm was first tested with a dataset consisting of a 50x500 matrix with random Gaussian numbers. The dataset was reduced and reconstructed, and the Root Mean Squared Error was calculated for every possible value of *q*(the dimensionality of the reduced dataset). The following figure illustrates the relation between the RMSE and the dimensionality:



As it can be expected, for random data, the more dimensions used in the reduced data, the lower the RMSE, until reaching a value close to 0 error for 50 dimensions(the original dimensionality of the dataset. One could think that using 50 dimensions would produce an error value equal to 0, but this is not the case due to the data lost during the reduction-reconstruction process.

Later, the same PCA algorithm was tested on a real dataset, the “*choles all*“ dataset, which is included by default in MATLAB. This is a dataset containing 21 numerical variables related to cholesterol levels.

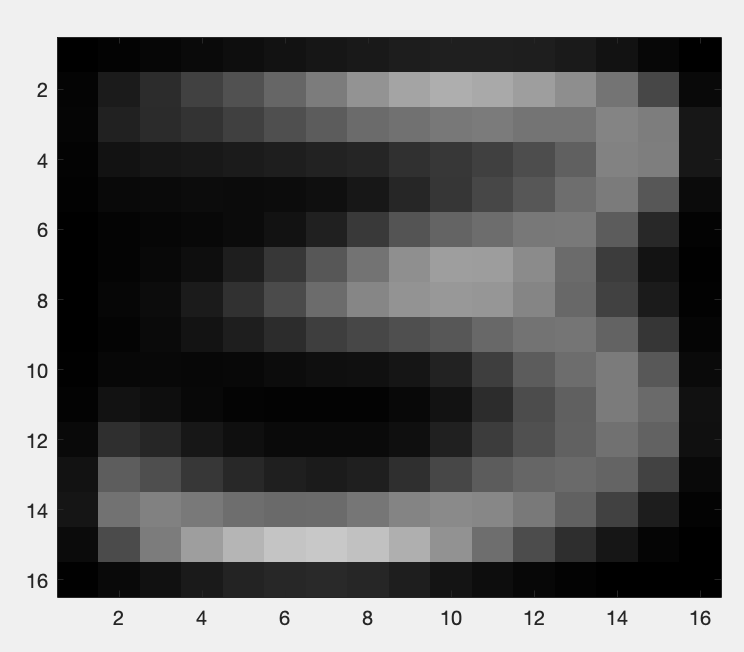
We applied PCA for all the possible dimensionality reductions and plotted the RMSE evolution:



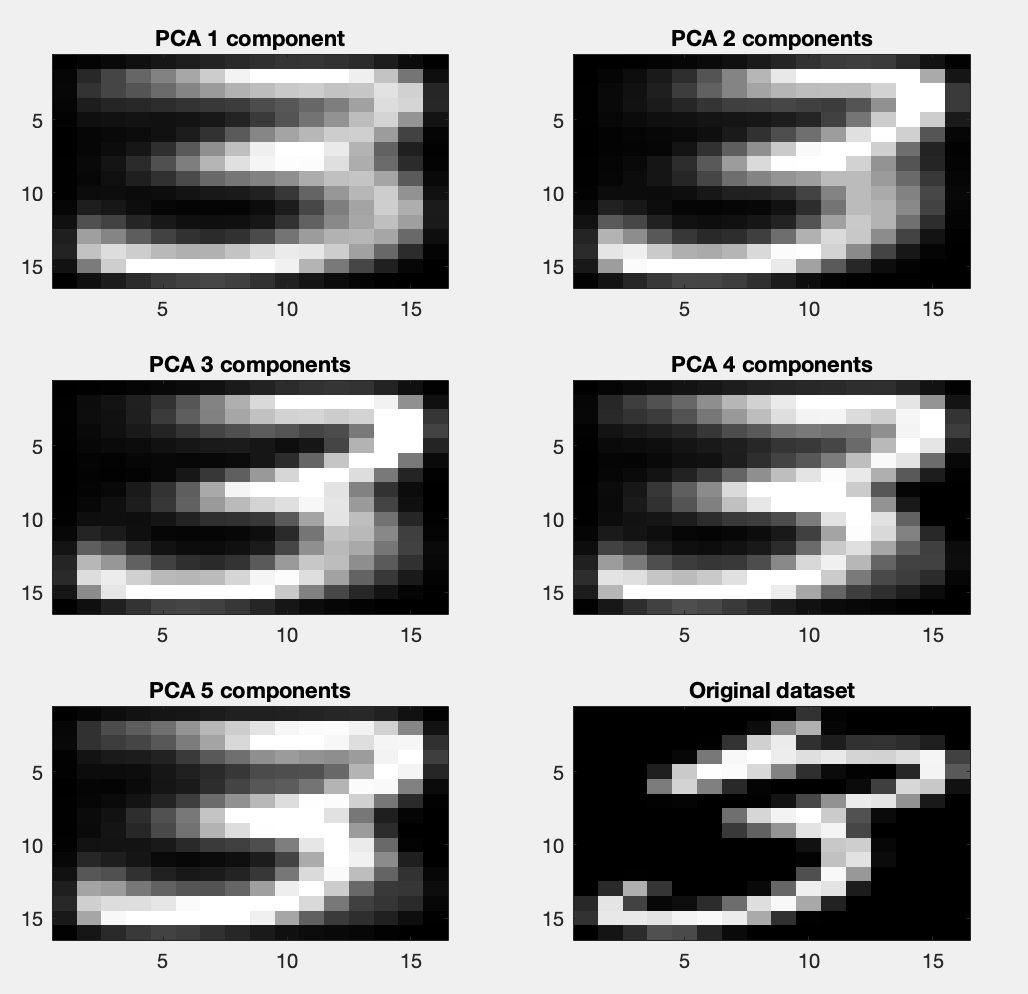
Unlike the random dataset, the RMSE values do not follow a straight downward line. This time, the lowest error value is achieved with a dimensionality of q=2. It can be seen that with highly correlated data, some of the dimensions contain most of the variance present in the data, so the original dataset can be greatly reduced without losing a lot of accuracy.

### PCA on handwritten digits

The second exercise on PCA consists on reducing the dimensionality of a dataset which contains images of the handwritten digit 3. This in another dataset which is included in MATLAB. By using the mean function, we first plot the representation of the average 3 contained in the dataset:

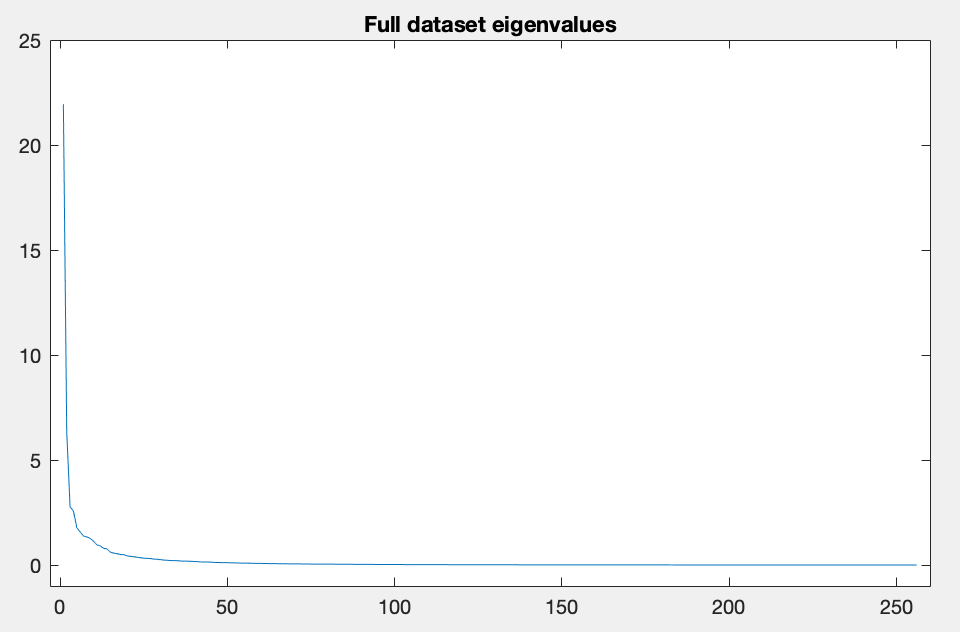


Then, to see the effect of some PCA reductions, we plot the resulting visualization of the reconstruction values given by PCA with one, two, three, four and five principal components. The original data is also included in the plot for comparison.



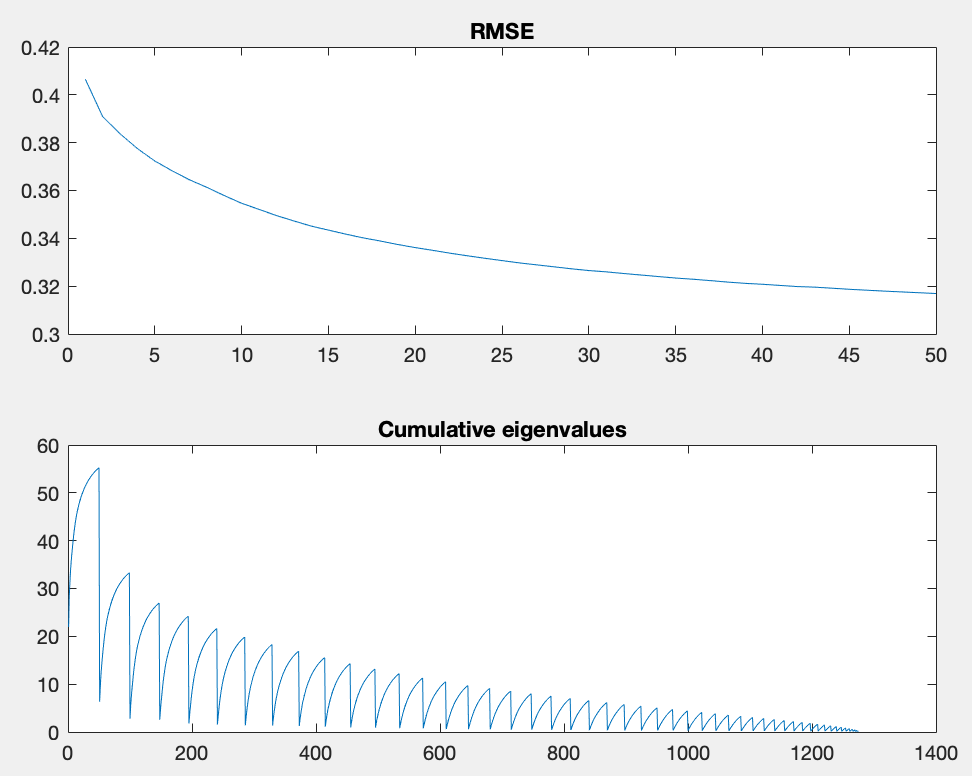
As it can be seen, even using a single principal component we can reconstruct a meaningful image.

We now plot the 256 eigenvalues, or principal components, of the full dataset:



It becomes obvious bu looking at the figure that the first few eigenvalues accumulate most of the variance of the dataset. Therefore, performing PCA reduction with this number of dimensions should return a good results, as we have showed in the previous comparison.

As a final exercise, we calculate the RMSE in function of the number of dimensions; and also the cumulative eigenvalues vector. This vector consists of the the sum of all but the *i* largest eigenvalues, for values from i=1 to i=50. If these two curves are plotted together, it can be seen that they follow a similar trend:



This relationship between both curves implies that the RMSE value of the reconstructed PCA data is proportional to its eigenvalue. Thus, this fact explains why projecting to the first few eigenvalues gives a small value for the rror if the eigenvalues fall off quickly.

## Stacked Autoencoders

# Generative models