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 dimensionality 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 a 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 trade off. This is basically avoiding over fitting the model with a specific dataset and rendering it unable to generalize. One way of achieving 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 consists 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.

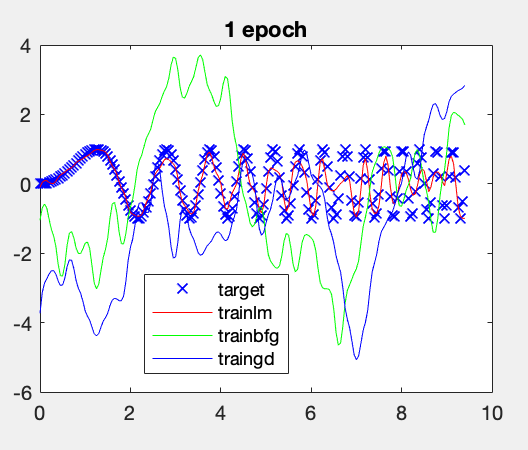
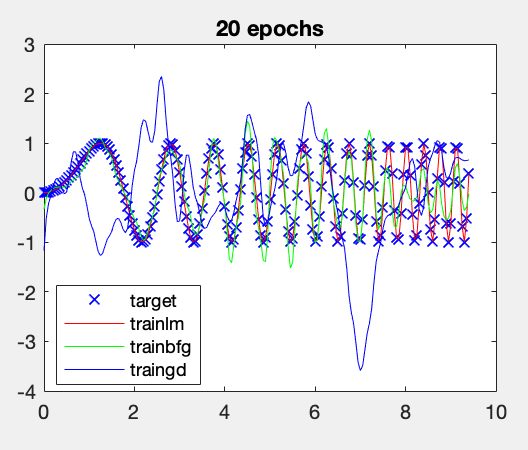
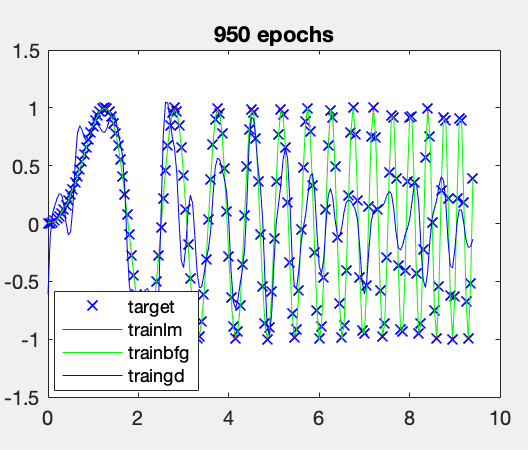
It is important to note that many other learning methods exists, such as Resilient Backpropagation or One Step Secant, but they have been left out of this analysis for the sake of clarity.

Figure 1. Comparison between trainlm(left), trainbfg(center) and traingd(right) with a different amount of epochs.

As we can see in Figure 1, 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, in Figure 2 we show the comparison of the linear fit achieved by LM and GD with 950 epochs.

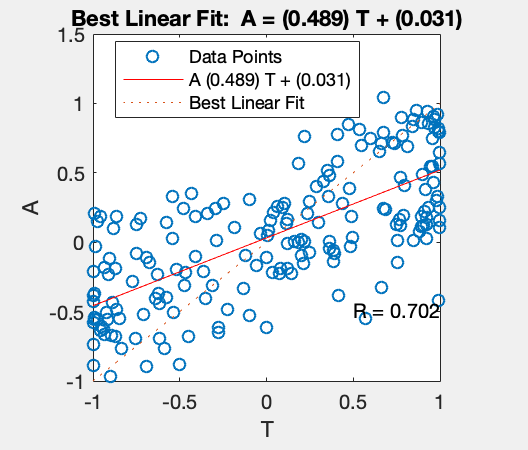
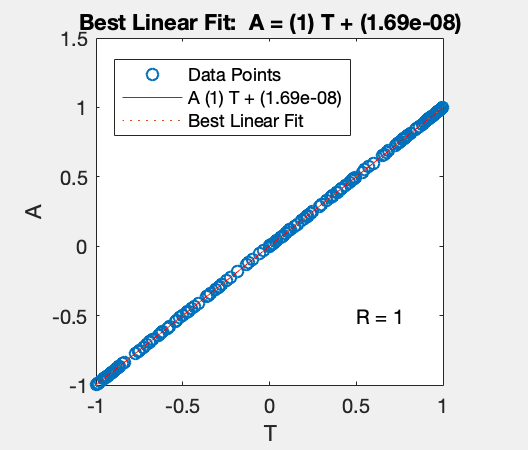
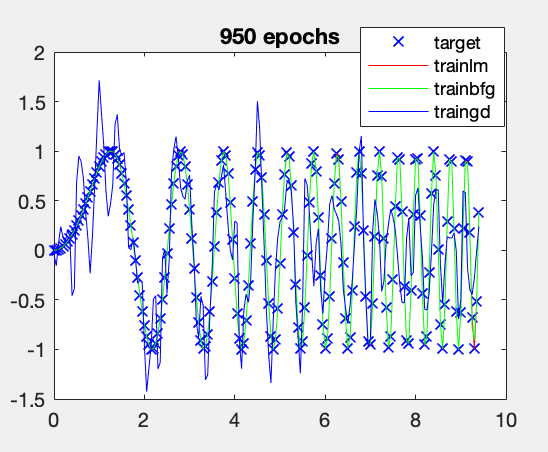


Figure 2. Linear fir comparison between trainlm(left) and traingd(right).

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.

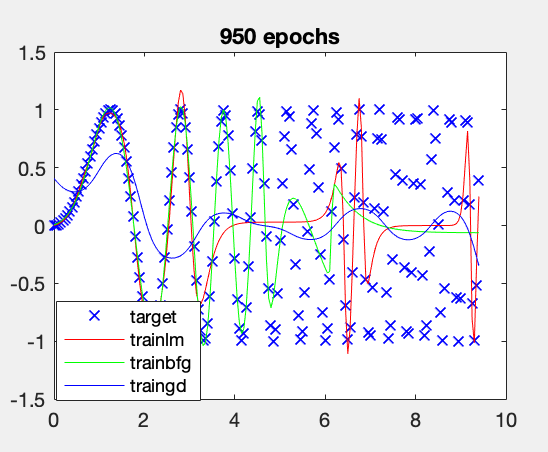


Figure 3. Effect of the number of hidden units. 10(left) vs 100(right).

### 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 of 5% has been included.

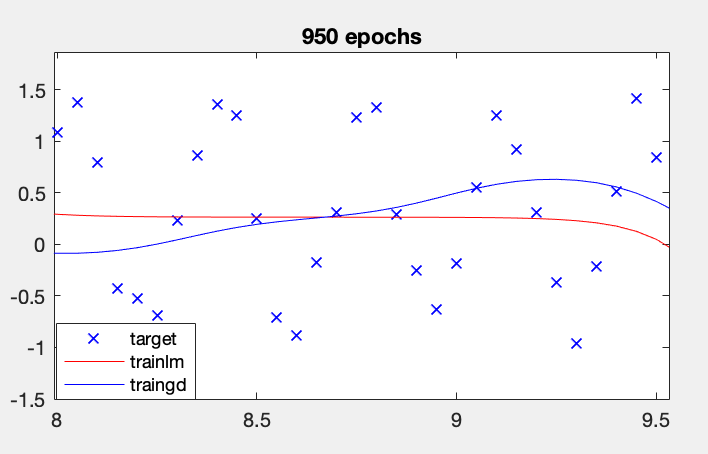
****

Figure 4. Effect of noisy data.

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 with the generated 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 existing 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 Figure 5.

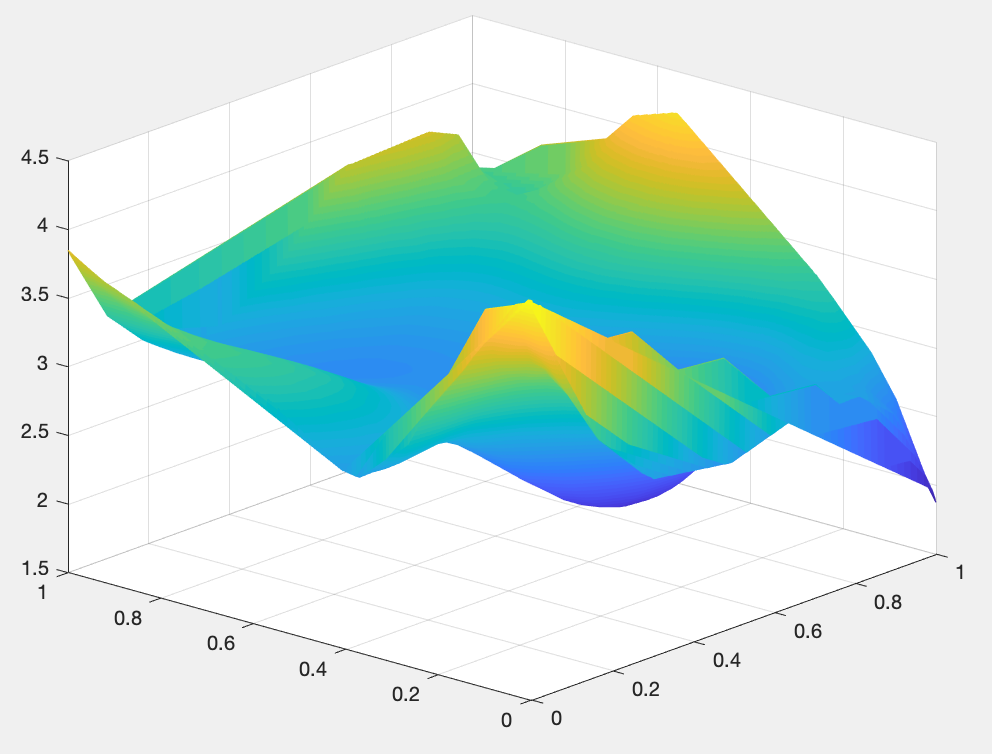


Figure 5. Mesh plot of the training dataset.

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.

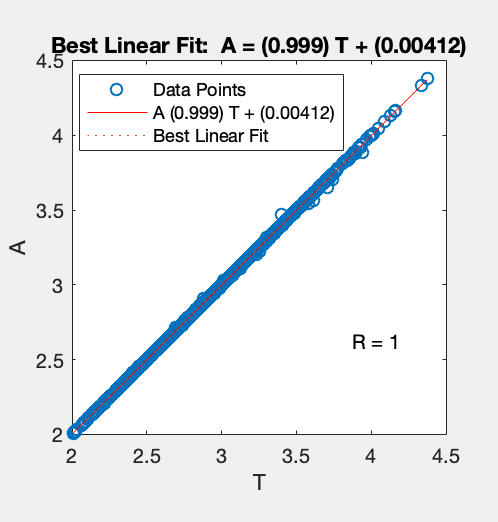


Figure 6. Linear fit of trainlm.

In Figure 7 we can see the actual test points vs. the prediction of the model.

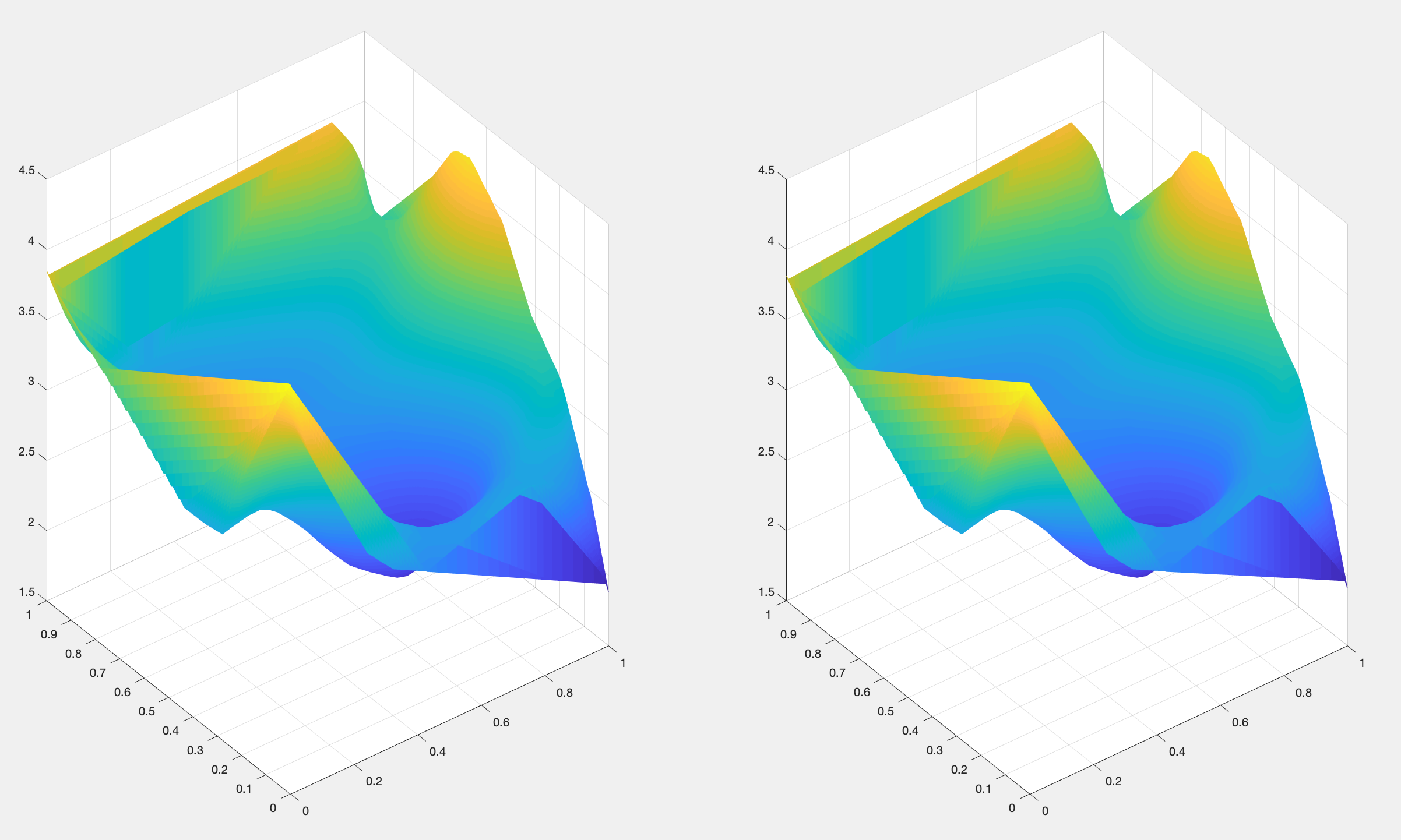


Figure 7. Training data vs. prediction.

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 be 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 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. TODO(fix this chart by increasing data points)

|  |  |  |
| --- | --- | --- |
| Random noise | trainlm(100 units, 950 epochs) | trainbr(100 units, 950 epochs) |
| 5% | 0,913 | 0,916 |
| 10% | 0,953 | 0,942 |
| 15% | 0,941 | 0,913 |

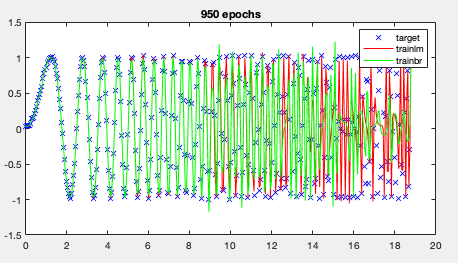


Figure 8. trainlm vs. trainbr.

# 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. Table 1 shows 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] |

Table 1. Inputs and corresponding attractors.

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.

### Highly symmetrical 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. This is due to the high symmetry of the input points.

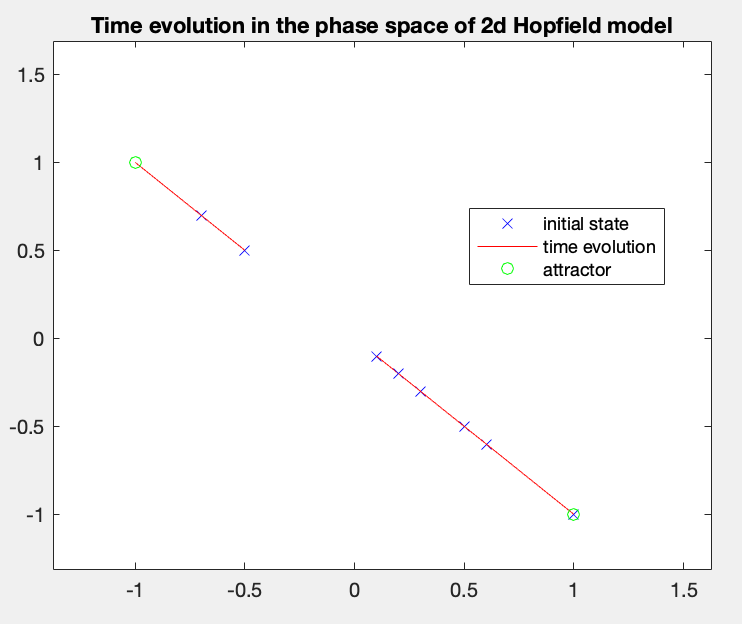


Figure 9. Time evolution of higly symmetrical points.

### 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.

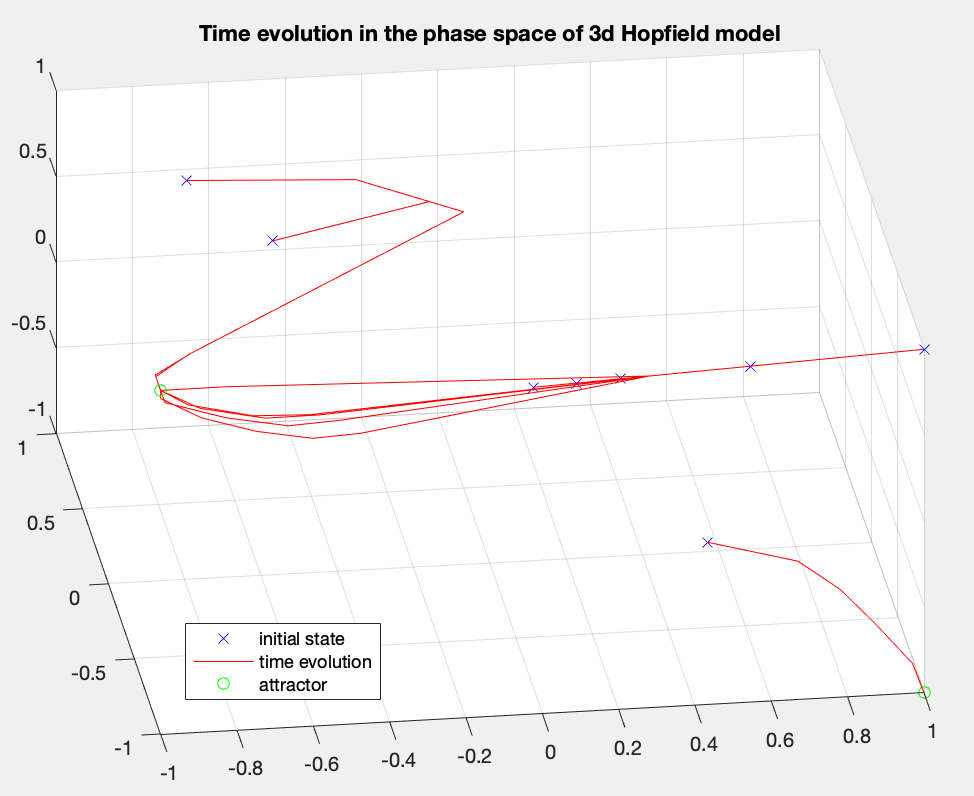


Figure 10. Time evolution of the 3-neuron Hofield network.

### 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. Figure 11 shows the predictions given a noise level of 5 and 1000 epochs.

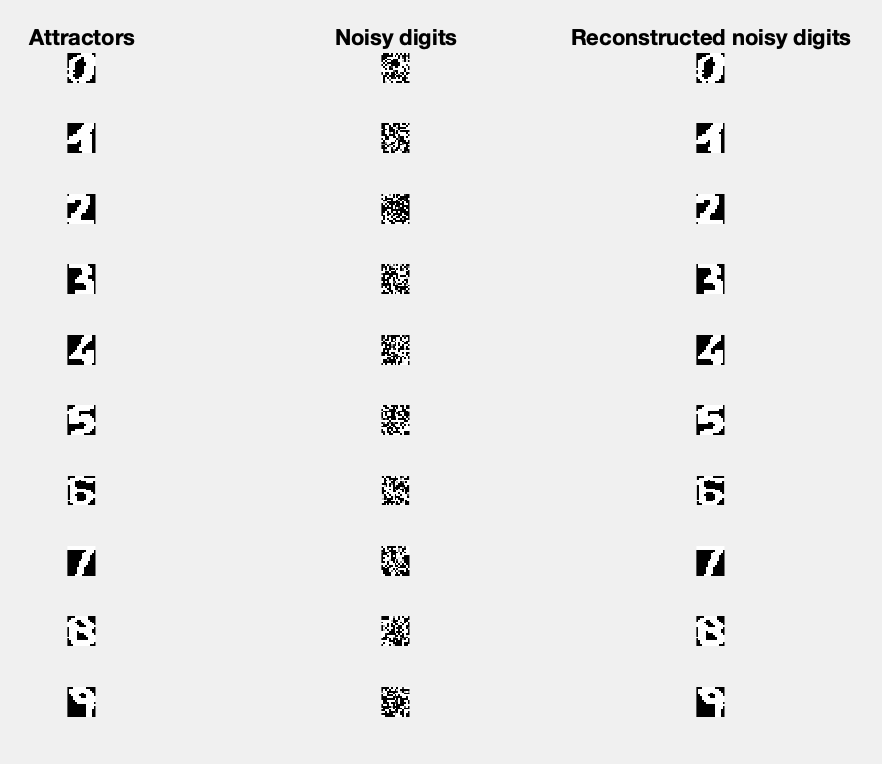


Figure 11. Digit prediction with noise level 5.

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).

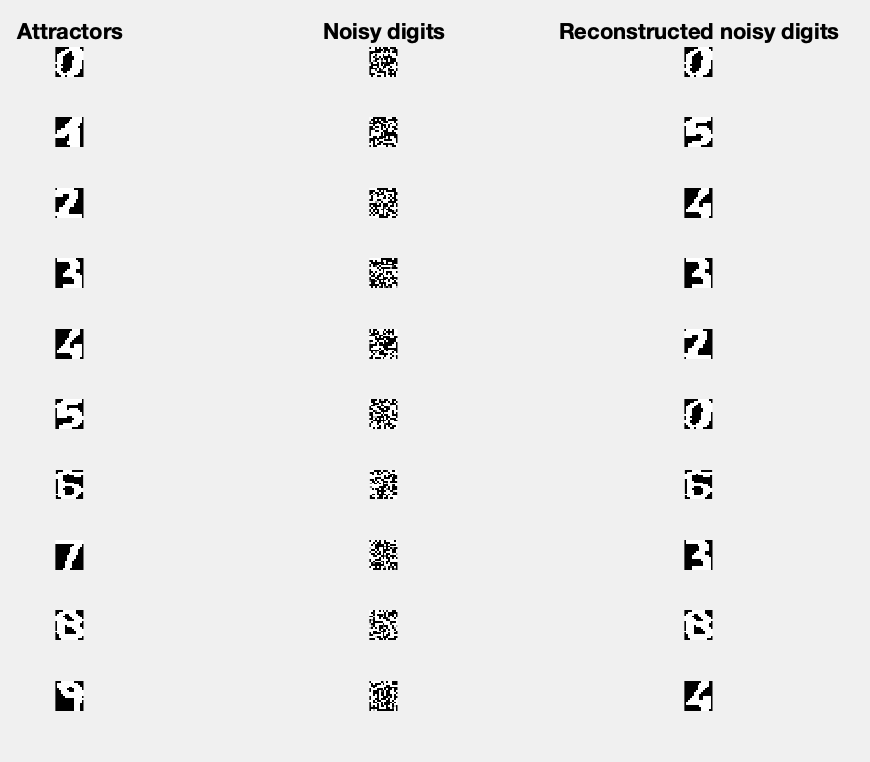


Figure 12. Digit prediction with noise level 9.

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

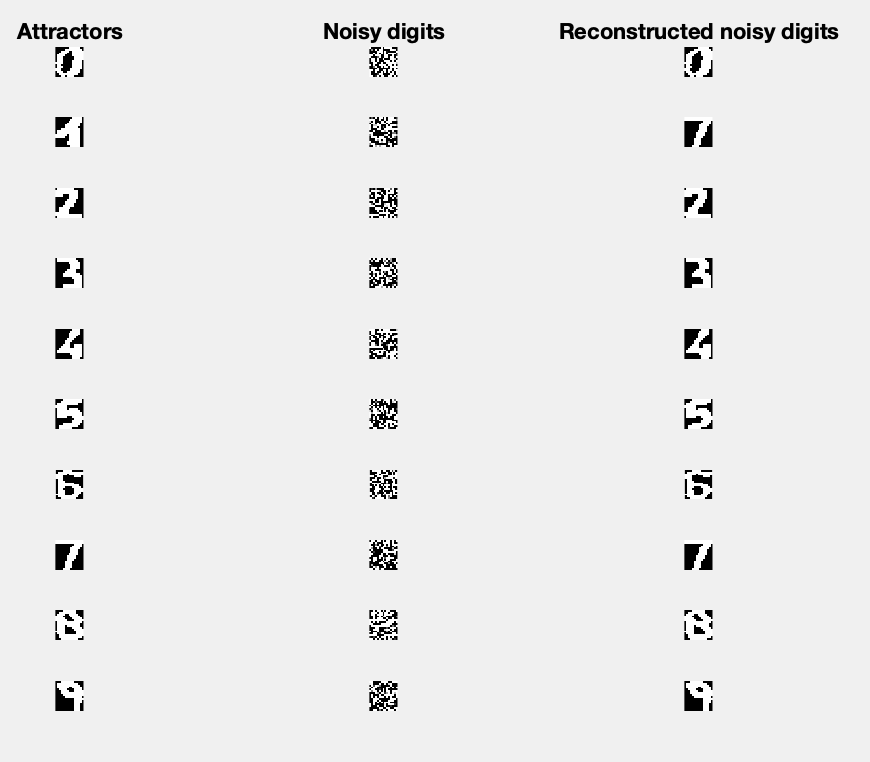
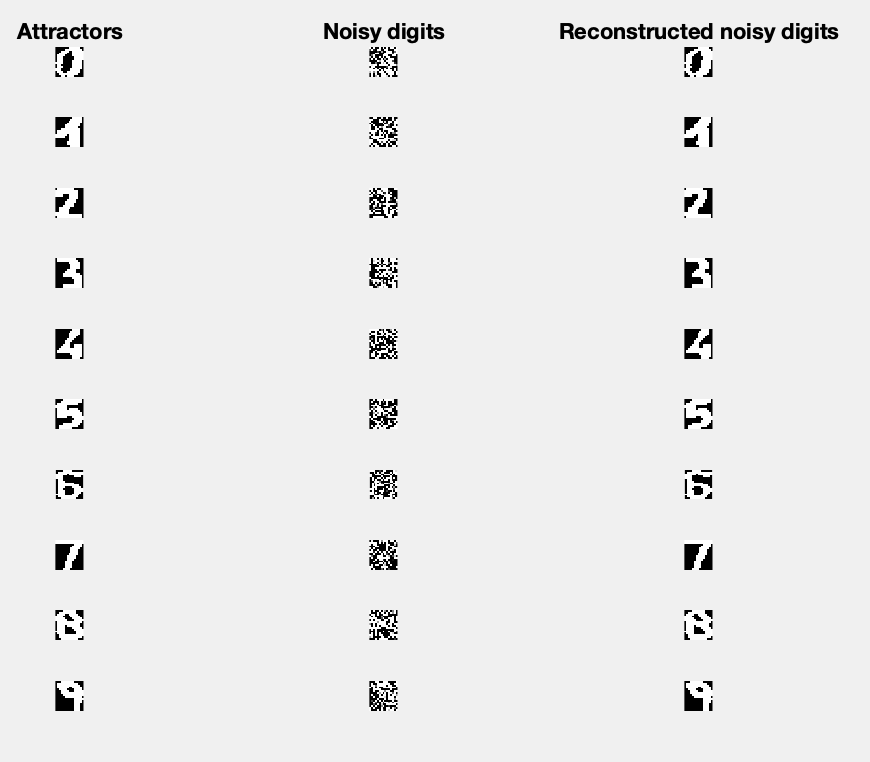


Figure 13. Digit prediction with noise level 5. 100 epochs(left) vs. 1000 epochs(right).

## 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 bit better, although takes much longer to train. This is expected, as trainbr generalizes better.

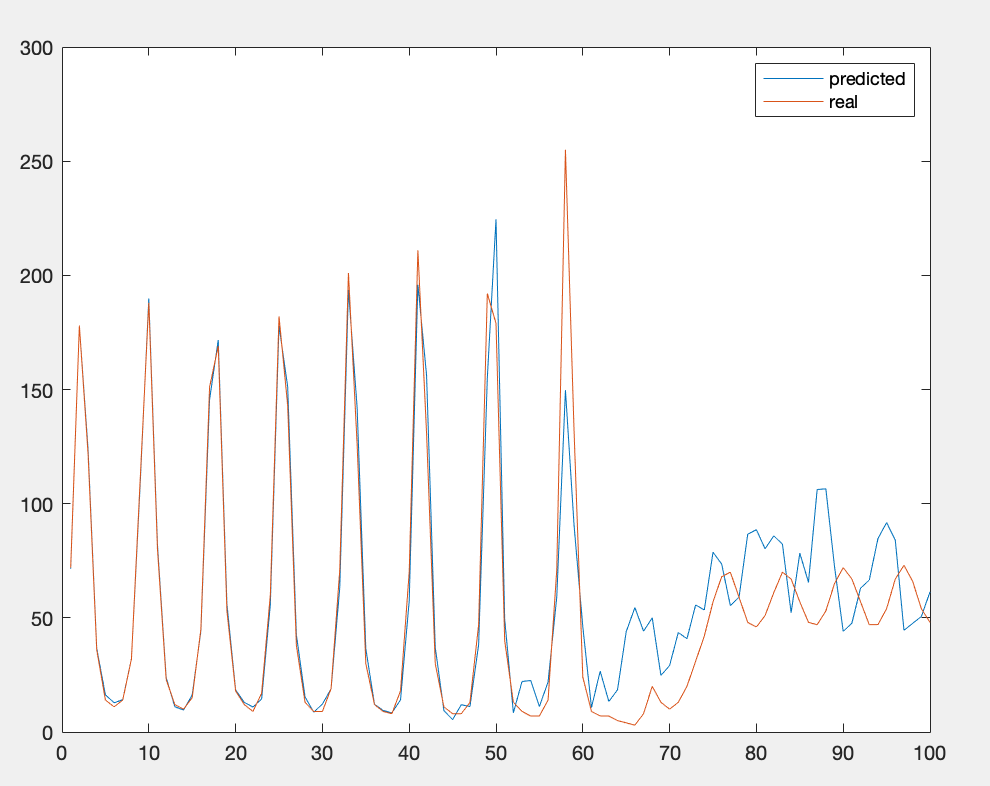
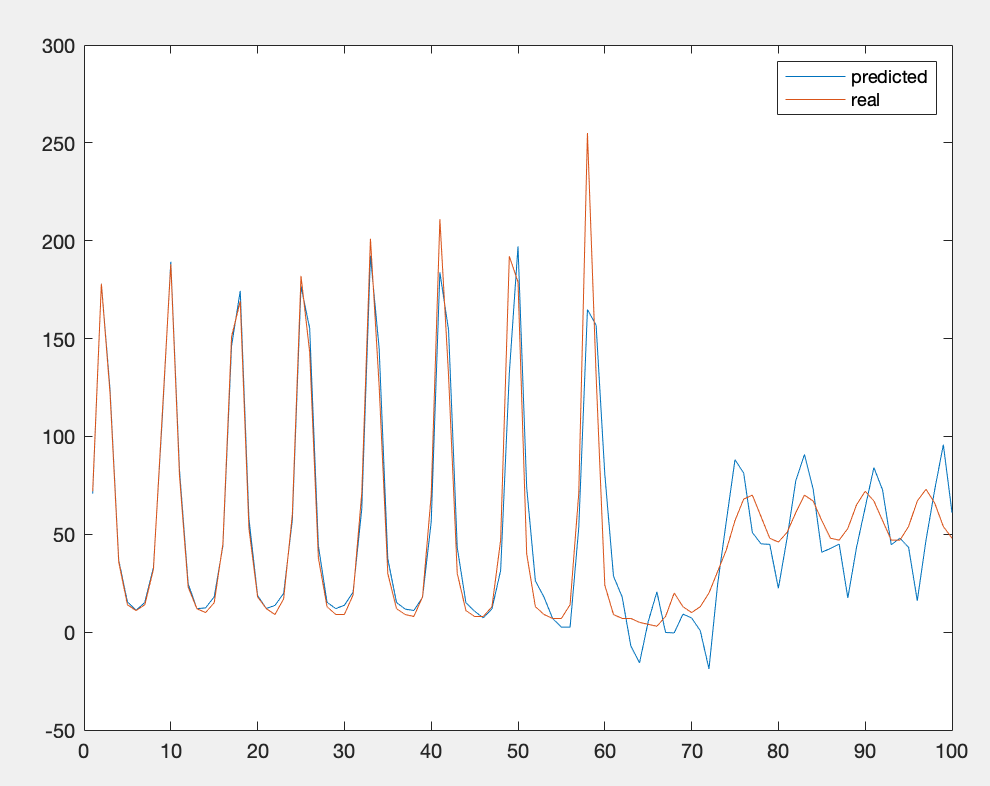
In the end, the best combination of training parameters was 40 hidden units, 1000 epochs and a lag value of 100. In Figure 14 we can see the best predictions obtained with these parameters and the trainlm(left) or trainbr(right) algorithms.

Figure 14. Comparison of trainlm and trainbr for time-series prediction.

### 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 variability 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 run a number of times, and the 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 Figure 15, which compares RMSE vs lag.

Figure 15. 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 Figure 16.

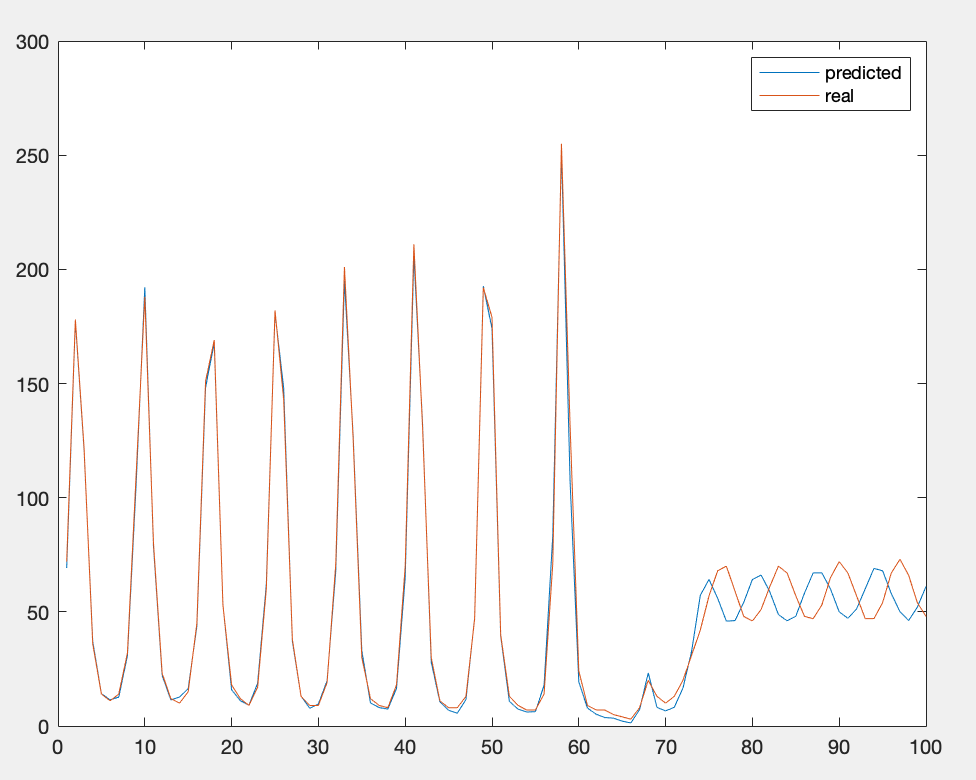


Figure 16. Best prediction.

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.

### 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.

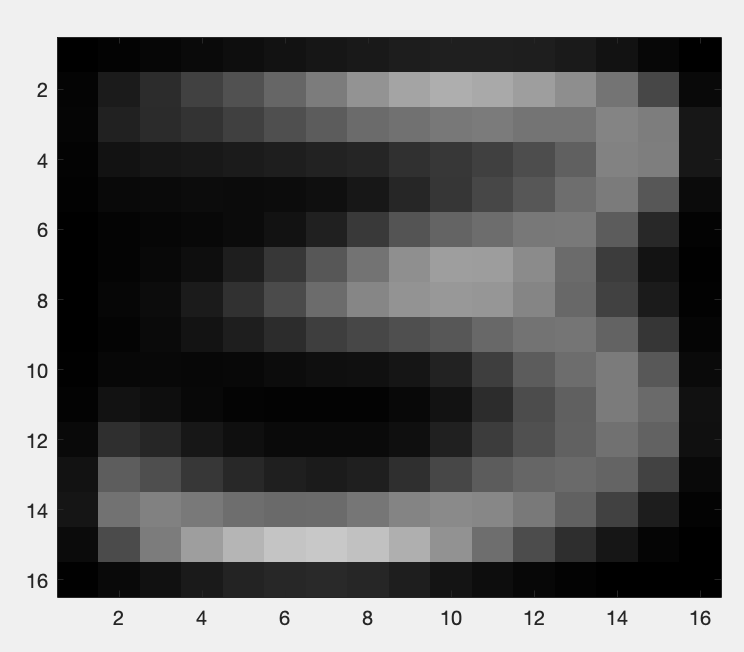


Figure 17. Average digit 3.

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.

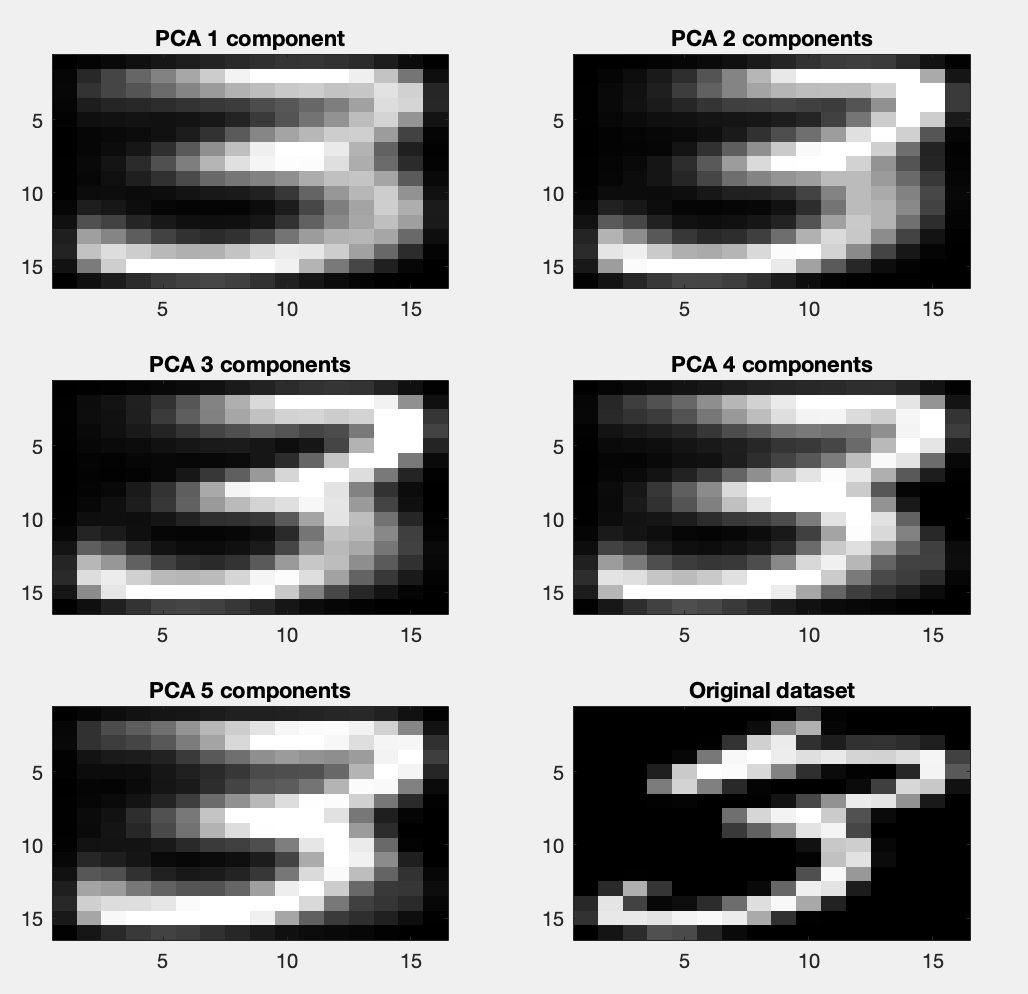


Figure 18. Reconstructed digits with 1, 2, 3, 4 and 5 principal components.

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.

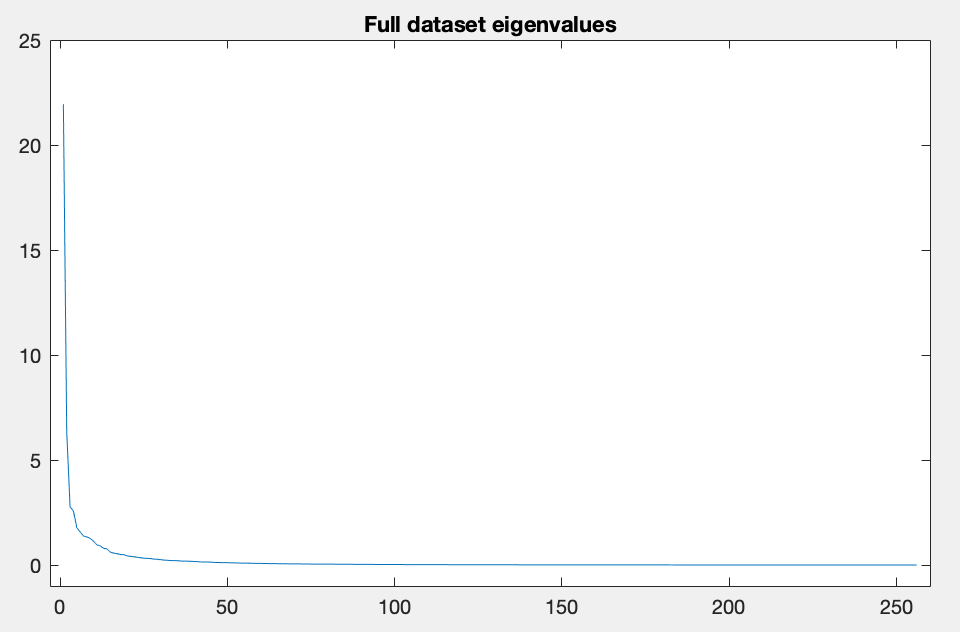


Figure 19. Principal components.

It becomes obvious by 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 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. They are shown in Figure 20.

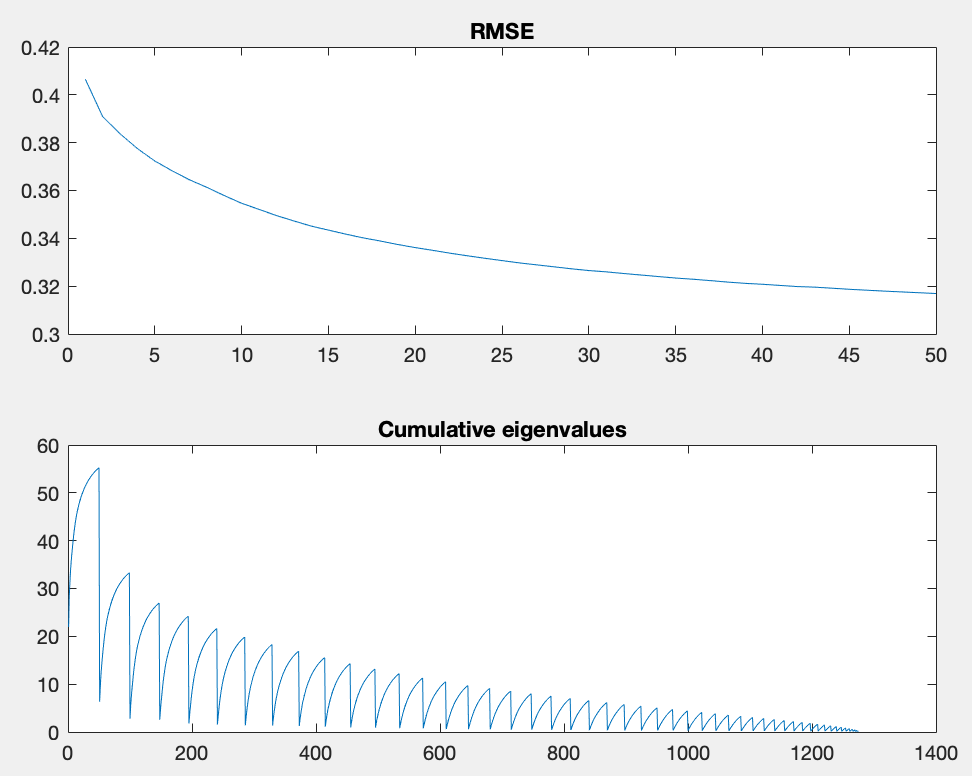


Figure 20. RMSE vs. cumulative eigenvalues.

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.

On a final note, one could think that using 256 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.

## Stacked Autoencoders

The next exercise consists on building a digit classifier using stacked autoencoders as the network architecture. Stacked autoencoders are a type of neural network which uses several autoencoders for its layers. Autoencoders are a type of neural network which uses backpropagation and sets the target values to be equal to the inputs, this means that it tries to learn an approximation of the identity function. The part from the input to the hidden layer is the encoder, and the part from the hidden layer to the output is the decoder.

The advantage of these networks is that some constrains can be applied, for example, limiting the number of hidden units in the intermediate layers. Doing this effectively means compressing the data, in a similar way as the PCA algorithm. In fact, autoencoders tend to learn a low-dimensional representation of the input data.

In this exercise, we will compare the performance of some stacked autoencoders vs. a normal multilayer neural network. For this, the digits dataset will be used, which contains images of single digits.

The architecture used for the network consists of two stacked autoencoders, the first one with 100 hidden units and the second one with 50, which are connected to a final softmax layer which classifies its input into any of the 10 possible digits to identify. As it can be seen in Figure 21, we are effectively reducing the input dimensionality from 784 to 50 dimensions.

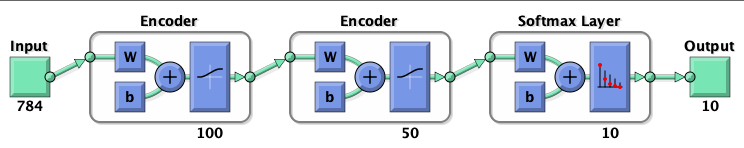


Figure 21. Architecture of the stacked autoencoder.

We can take a look to the weights of the autoencoders to see what patterns they are learning. For example, Figure 22 shows the weights of the first autoencoder after training.

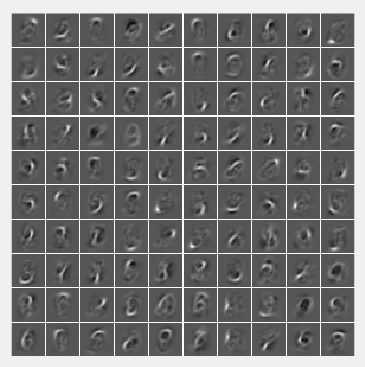


Figure 22. Visualization of the weights of the first autoencoder.

We can observe how some units clearly resemble to some specific digits such as 6, 9 or 5. This means that these units are specializing in recognizing this specific digits, thus giving more importance to them by outputting a bigger numerical value due to its weight.

The accuracy score of this network is 83.12%, but this can be vastly improved by fine-tuning the network. So far, the predictions have been performed in an unsupervised way; the network only had access to the input data, without the corresponding labels.

Fine tuning consists on performing backpropagation on the whole multilayer network by retraining it on the training data in a supervised way, i.e. with the data labels. After performing these fine-tuning step, the accuracy score jumps up to 99.74%, an almost perfect result, which highlights the importance of fine-tuning, and the potential of stacked autoencoders.

The results can be seen on the confusion matrix in Figure 23.

We can observe how several digits are classified with 100% accuracy on the test set, and all of them have accuracy scores over 99%, with the lowest score being 99.2% for digit 1.

But why should autoencoders be used instead of multilayer neural networks? To answer this question we build several networks using different number of layer, epochs and hidden units. In summary, the fine tuned stacked autoencoders perform better than every other neural network combination.

Figure 23. Confusion matrix.

We first tried a network with a single hidden layer, using the MATLAB *paternnet* command. The architecture can be seen in the following figure:

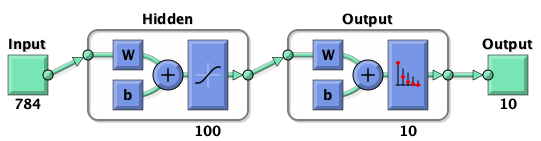


Figure 24. patternet architecture.

As this kind of networks are not deterministic(because the weights are randomly initialized), we ran trained the network 10 times and averaged the resulting accuracy. After this 10 runs, the average accuracy of the network was 96.79%.

The same test was repeated for networks of 2 and 3 hidden layers. Their respective accuracies were 96.85% and 97.18%. One can notice that even though accuracy is slowly increasing, it is never as good as it was with the stacked autoencoders. Increasing the number of layers comes with an exponential increase in the computational cost, and autoencoders perform better with 2 layers than multilayer networks do with 3+. Therefore we can say that stacked autoencoders are better suited for this kind of classification tasks.

Several other parameter combinations were tested, but all of them yielded worse results.

## Convolutional Neural Networks

Convolutional Neural Networks(CNNs) is a deep learning architecture for neural networks in which the concept of local connectivity is put into practice. Unlike normal multilayer neural networks, CNNs do not connect all nodes from layers which are next to each other. The idea is that points that are closer together have the potential of being much more connected than those which are far apart from each other. In image analysis this would translate to pixels which are closer together having a bigger impact on each other than pixel which are more distant, i.e. pixels which are close may represent the same object, while distant pixels might represent a different object.

CNNs are able to capture the spatial dependencies between pixels while learning directly from the raw matrix of pixel values. In normal neural netoworks this data must be preprocessed into a vector, which increases the preprocessing costs. CNNs reduce the raw images into a lower dimensionality by grouping pixels which are close together bu using a kernel function. For example, the original image can be analyzed by grouping pixels in a 3x3x1 matrix with weights, and applying this kernel to the whole image.

By doing this, the network is effectively extracting high-level features, much like PCA and autoencoders. The difference is that each layer in the CNN learns a different level of features, from low in the first layer(features color or edges) to high in consequent layers.

In addition to convolutional layers, CNNs also have a pooling layer which serves to reduce dimensionality even further. There are two kinds of pooling: average and maximum. Usually, maximum pooling performs much better given that it acts as a noise reducer.

Finally, these layers are in turn connected to a fully-connected layer in order to be able to learn non-linear combinations of features from the outputs of the convolutional layers. Then, as in the autoencoders, we can plug in a softmax classification layer to predict digits.

### CNNex exercise

Running the MATLAB-provided *CNNex* exercise, one can take a look at how to:

* Preprocess input data
* Download CNN weights pretrained on the Imagenet dataset using Alexnet as architecture
* Visualize layer weights
* Train and evaluate a classifier based on these weights

The first thing we will do is take a look at the weights of the first convolutional layer, this can be done by accessing the *Layers* property of the CNN object which contains the pre-trained network:

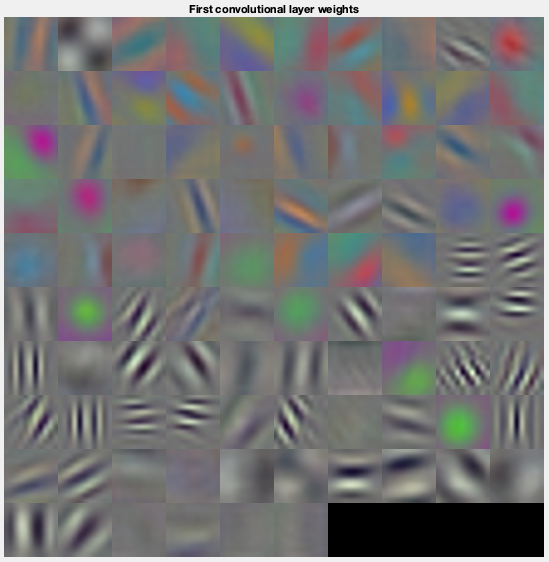


Figure 25. First convolutional layer weights.

As we know, the weights of the first convolutional layer represent low-level features such as colors and edges. We can observe in the image how these weights have very abstract and different shapes and colors. It is important to note that the input dimensions were 227x227, while the first convolutional layer has just 96 weights. This is how CNNs reduce data dimensionality.

One can also take a look at the first 5 layers of the network. Specifically, they are:

1. ImageInputLayer
2. Convolution2DLayer
3. ReLULayer
4. CrossChannelNormalizationLayer
5. MaxPooling2DLayer

Out of these 5 layers, we know that the ReLU and Cross Channel Normalization layers do not modify the dimensionality of data. Thus, we have two layers which modify dimensionality: the Convolution and the Pooling layers.

The first convolution layer applies a 11x11 filter, thus, we can calculate that this filter would be applied 55 times vertically and horizontally on the original 227x227 images. The output dimensions of this layer are then 55x55x96. Later, the pooling layer has a pool size of 3x3 with a stride of 2x2. Knowing these parameters we can calculate again the output dimensions, which in this case would be **27x27x96.**

These calculations have been made using the following formulas:

Convolution: 1 + 4(i - 1) + 11 - 1 = input-size

Pooling: 1 + 2(i - 1) + 3 - 1 = input-size

Where *i* is the output size of each layer.

Finally, one could calculate the dimensions of the inputs before the classification part, i.e. after all the convolution and pooling layers. By taking a look at the network architecture, we can see that the dimensions of the first fully connected layer is just a vector of size 9216. This means that the network has effectively reduced the data dimensions from 227x227x3 to 9216, which translates to removing 145371 dimensions. This shows why CNNs are useful to avoid the curse of dimensionality, while maintaining a great accuracy due to the way that convolutional layers learn from data. A fully connected network would not be able to reduce dimensions and thus the computational complexity would increase exponentially.

### CNNDigits exercise

Finally, we will create and train a CNN capable of classifying handwritten digits, the same task that was performed by stacked autoencoders. Several combinations of parameters were tried. We will showcase here 2 of them, training for 30 epochs:

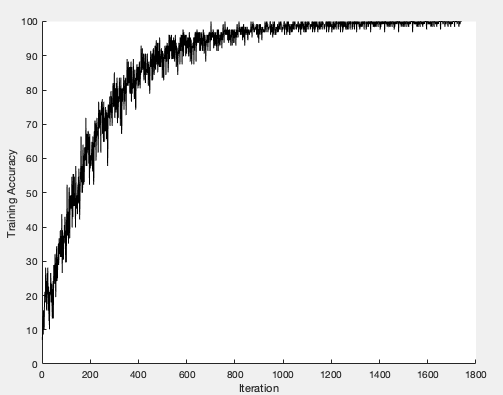


Figure 26. Learning curve of architecture 1.

* Architecture 1
  + imageInput(28, 28, 1)
  + convolution2D(5, 24)
  + relu
  + maxPooling2D(2, 2)
  + convolution2D(5, 36)
  + relu
  + fullyConnected(10)
  + softMax
  + classificationLayer

Accuracy: 97.68%

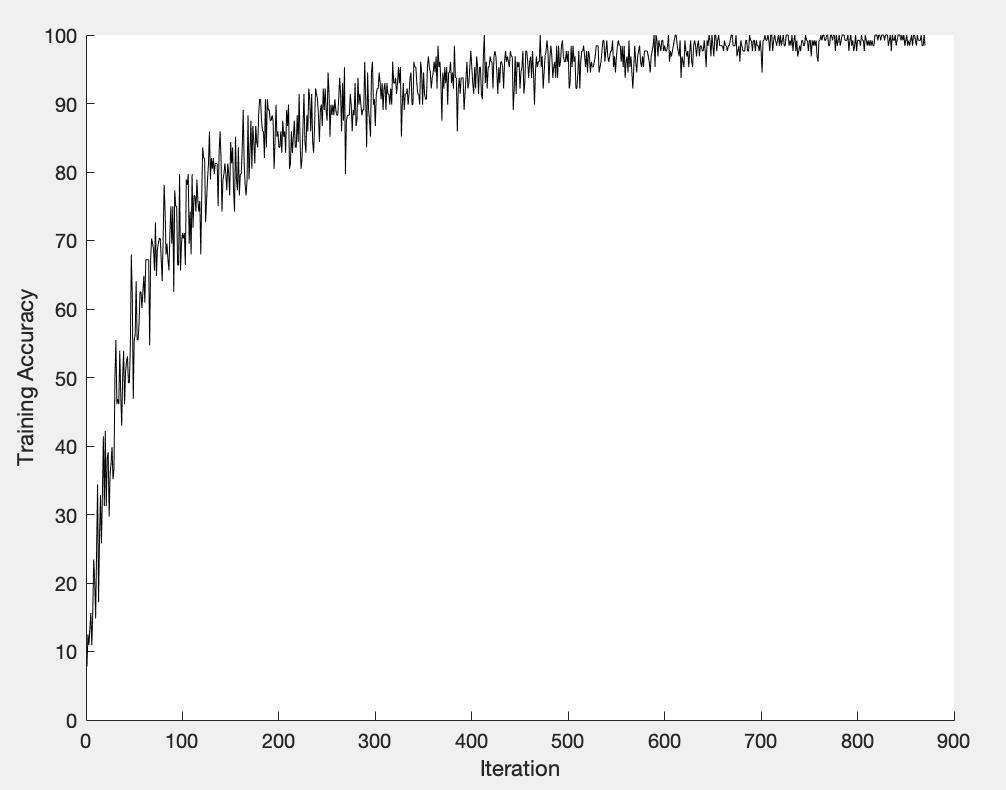
* Architecture 2

Figure 27. Learning curve of architecture 2.

* + imageInput(28, 28, 1)
  + convolution2D(5, 20)
  + relu
  + maxPooling2D(2, 2)
  + fullyConnected(10)
  + softMax
  + classificationLayer

Accuracy: 95.96%

As it can be seen, given enough epochs, both architectures converge to similar accuracy values, although the 1st architecture achieves a better score, due to the additional layer.

It is important to notice that for this task, stacked autoencoders performed significantly better, and are faster to train.

# Generative models

As it can be deduced by their name, Generative Models aim at learning the function behind the data distribution of a dataset in order to be able to generate new data that follows that underlying function. This function is not always possible to learn perfectly, so the different models try to approximate it as good as possible.

There are several options for Generative Model learning, so we will analyze and experiment with some of them.

## Restricted Boltzmann Machines

A Restricted Boltzmann Machine is a kind of neural network pertaining to the group of generative models, which tries to learn the probability distribution represented by the training data. After successfully learning this, a RBM provides a representation of the distribution underlying the observations, and thus can generate data points which follow this distribution by sampling. For example, it can be used to reconstruct missing parts of an image.

They are called restricted because there are no connection between pairs of hidden layers, nor pairs of visible layers.

### Training the model

In order to experiment with RBMs, we loaded a dataset consisting on images of handwritten digits and trained the model with this data. In Figure 28 we plotted some of the original images.

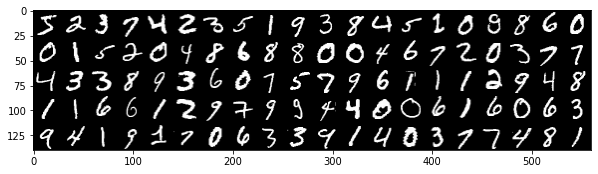
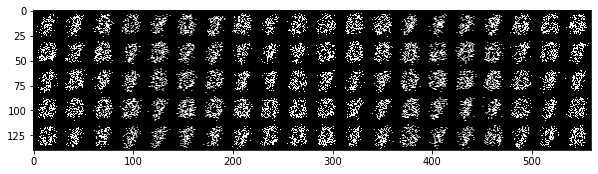
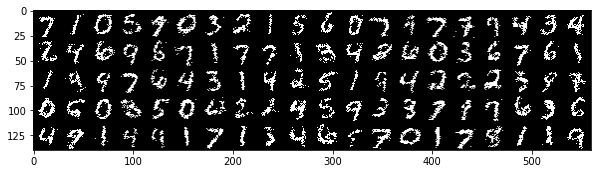


Figure 28. Some original images.

The RBM accepts 2 parameters whose effect we want to analyze: number of components and number of iterations. The following images were all generated using Gibbs sampling with 1 step.

 10 components 10 iterations 100 components 10 iterations

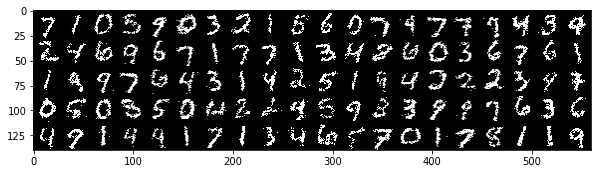
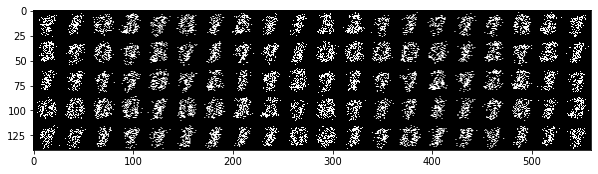
 10 components 30 iterations 100 components 30 iterations

Figure 29. Comparison of the effect of components and iterations.

With 100 components and 30 iterations the results are pretty good when compared to the original images:

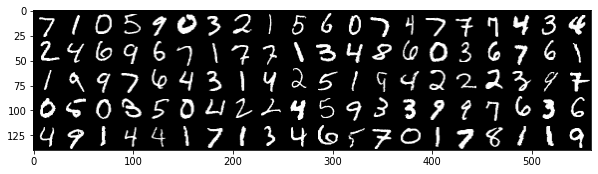


Figure 30. Predictions with 100 components and 30 iterations.

However, using so many components increases the computational complexity significantly. For example, 10 components take 4.4 seconds per iteration, whereas 100 components take 17.5 seconds per iteration. On the other hand, increasing the number of iterations improves the results slightly, but good results cannot be obtained without increasing the number of components.

Finally, Gibbs sampling works by sampling from the distribution n times, and then it takes the average. Thus, increasing the number of Gibbs steps does not improve the resulting samples, as it converges into showing the most frequent shape, in this case a mix of 7 and 1. According to some online documentation and papers, which we can corroborate, a single sampling step gives surprisingly good results.

### Reconstructing images

With the model already trained, we can use it to reconstruct unseen images which are missing a some pixel rows. In the following tests, 10 rows of pixels were removed(the images are 28x28), similar results were obtained after removing 15 rows, but removing 20 rows gives produces unreliable reconstructions.

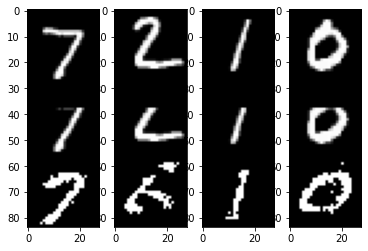
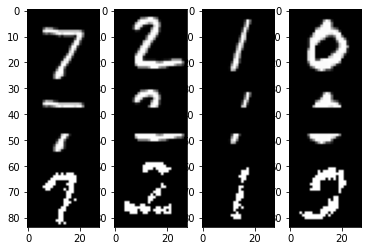
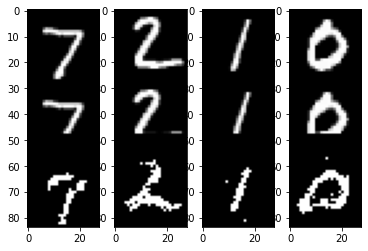
 Removing top rows Removing central rows Removing bottom rows

Figure 31. Comparison of removed rows. Original(top), incomplete(middle), predicted(bottom).

As we can see, the model does not provide perfect reconstructions, although the results are normally quite accurate.

## Deep Boltzmann Machines

Deep Boltzmann Machines(DBMs) are a series of RBM stacked on top of each other, much like the architecture of stacked autoencoders. There is full connectivity between neighboring layers, but none between the rest. First we will compare the components extracted by the previous RBM and the two layers of the pretrained DBM on the MNIST dataset. Results can be seen in Figure 32.

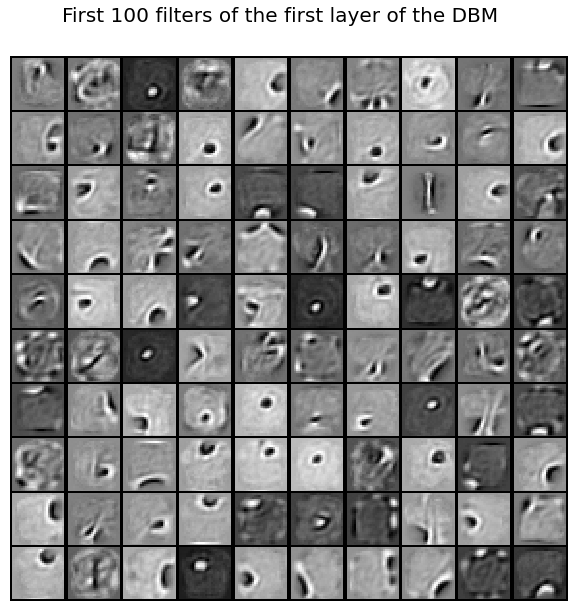
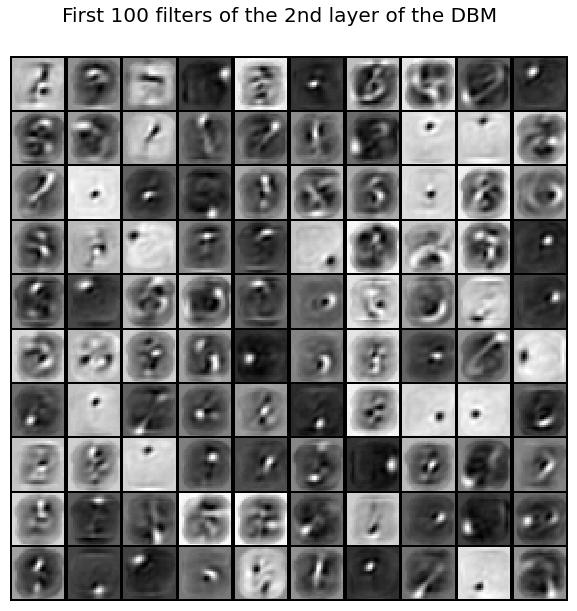
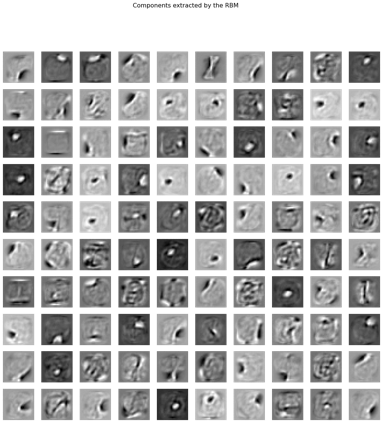
 Components extracted by the RBM

Figure 32. Comparison of extracted components.

It can be seen that the weights of the RBM are similar to the weights of the first layer of the DBM. These weights represent top-level knowledge, and thus look very abstract. They learn things like edges and shapes. The second layer is more low-level, and we can distinguish some shapes really similar to some digits. This layer is more specialized and it learns about the digits themselves. This behavior is similar to that of CNNs. Seeing these layers, one could expect much better accuracy from the DBM model.

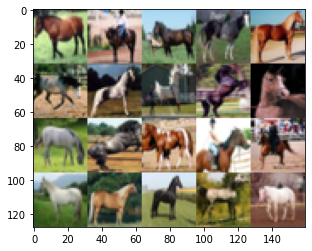
To test this assumption, some samples were generated in the same way as they were generated with the RBM model:



Figure 33. Generated samples.

It is really noticeable how much more precise these samples are. The line tracing is better defined are more realistic. There are less missing spots, and almost all generated images are lisible.

## Generative Adversarial Networks(GANs)

GANs are a special network architecture in which two separate neural networks are competing with each other. One of them, called *generator*, creates new data instances(e.g. images) and feeds this data besides some data points of the real dataset to the other network, called *discriminator*. The discriminator then tries to guess if the input data is real or fake, and assigns a probability to each instance. Finally, the generator updates its weights in order to increase the probability of fooling the discriminator.

We will train a DGAN architecture on the CIFAR dataset, which contains 32x32 color images from different categories. Specifically, we selected category 7, which corresponds to horses.

GANs need to be trained for a meaningful number of batches before they are able to produce realistic results. We trained the network for 20.000 batches, with a batch size of 32, and plotted the resulting predictions every 50 batches.

Figure 34. Images from the horse class.

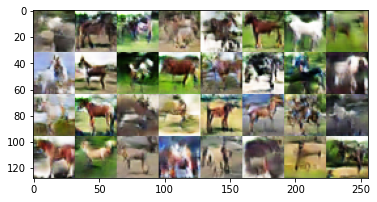
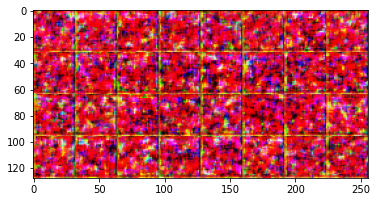


Figure 35. Generated images after 1(left) and 20.000(right) batches.

Although some of the generated images are blurry or distorted, we can observe how, generally, results are good and realistic when compared to the original dataset.

When monitoring the loss and accuracy values of both networks, we obtain the following data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch | Acc D | Loss D | Acc G | Loss G |
| 1 | 0.453 | 1.116 | 0.812 | 0.432 |
| 5.000 | 0.578 | 0.697 | 0.297 | 0.883 |
| 10.000 | 0.469 | 0.713 | 0.375 | 0.761 |
| 15.000 | 0.578 | 0.681 | 0.344 | 0.790 |
| 20.000 | 0.625 | 0.676 | 0.281 | 0.843 |

Table 2. Comparison of losses and accuracies.

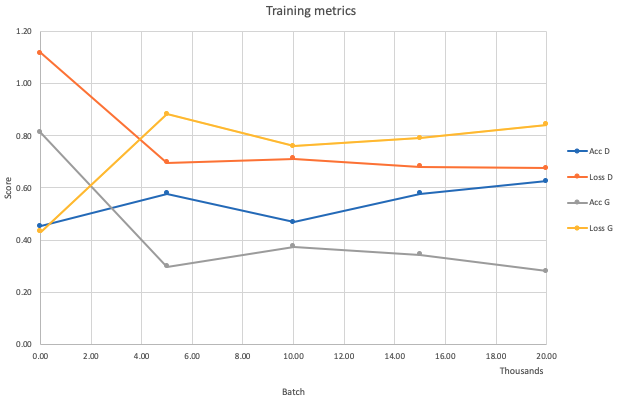


Figure 36. Plot of the accuracies and losses of the DCGAN.

We can observe an increasing trend in the discriminator accuracy, whereas the generator accuracy decreases. As expected, the losses are inversely proportional to the accuracies, so if the accuracy increases, the loss decreases.

Talking about the stability of the training, one can observe that the resulting generated images after 20.000 batches are realistic and varied. The “mode collapse” phenomenon has not been found during the training of the networks on this dataset. As opposed to what happens with the mnist dataset.

## Optimal Transport

Optimal Transport(OT) theory studies the properties of the least costly transport of one thing from one point to another as well as its efficient computation. It is used to solve the assignment problem, i.e. how to minimize the cost of transporting good from a factory to several stores based on the demand of the stores and the production quantity of the factory.

In practice, OT is often times not used to obtain the OT map, but to find the statistical divergence of two distributions, i.e. measuring dissimilarity between the distributions.

We will use two similar images, which have different colors, to train the OT algorithms(EMD and Sinkhorn). Then we will use the resulting models to transform image 1 into image 2 and vice versa.

The extracted colors can be seen in Figure 37.

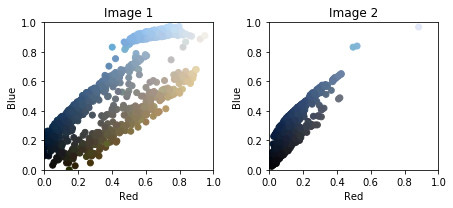


Figure 37. Extracted colors.

Figure 38 shows the input images next to the resulting transformations.

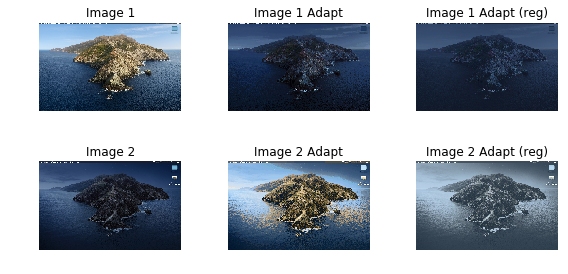


Figure 38. Image transformations. Original(left), EMD(middle), OT(right).

We can see that EMD gives better results, including lightning. In this particular case, pixel swapping would have produced a good result, as the only difference between images is the color, not the position. But OT allows for optimal color swapping of images with different objects in them.

### Wasserstein GAN

As a final exercise we will compare the quality of predictions made on the mnist dataset by a GAN, a Wasserstein GAN with weight clipping and a Wasserstein GAN with gradient penalty. The Wasserstein GANs use the Earth Mover Distance or Wassertein distance as loss function for the networks.

All these models have a relatively simple architecture, so they will inevitably produce worse results than the DCGAN that was analyzed earlier. The models were trained for 20.000 epochs.

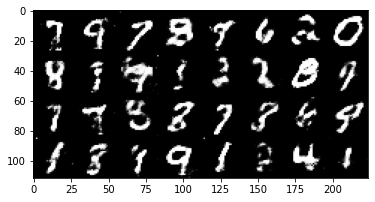


Figure 39. Generated digits. GAN(left), WGAN-WC(middle), WGAN-GP(right).

The results after 20.000 are pretty similar for the three models, although weight clipping produces more lisible results. Using Wasserstein as loss function results in better stability, as generated samples do not converge to a unique solution. However, we could not determine a clear increase in quality.

Here we can see the final training metrics of the three models:

Batch 20000, D loss: 0.505 D acc: 0.7422 G loss: 1.4689 G acc: 0.0703

Batch 20000, D loss: -0.0006 D acc: 0.0 G loss: 0.0038 G acc: 0.0

Batch 20000, D loss: -0.1026 D acc: 0.0 G loss: -0.1752 G acc: 0.0

Analyzing accuracy values does not make sense for the 2 Waserstein GANs, as the discriminator does not output a class, but a real value. In addition, one can see that some loss values are negative, and this is completely fine as the function that the model is trying to maximize is the distance between the fake and real batches for the discriminator, not the loss.