Artificial Neural Networks and Deep Learning

Assignments
Session 1: Supervised learning and generalization
Session 2: Recurrent neural networks
Session 3: Deep feature learning
Session 4: Generative models

Alexander J. Lindhardt
Student number: r0826077

KU Leuven
May 25, 2021

Contents

1 Exercise Session 1: Supervised learning and generalization

1.1 Function approximation

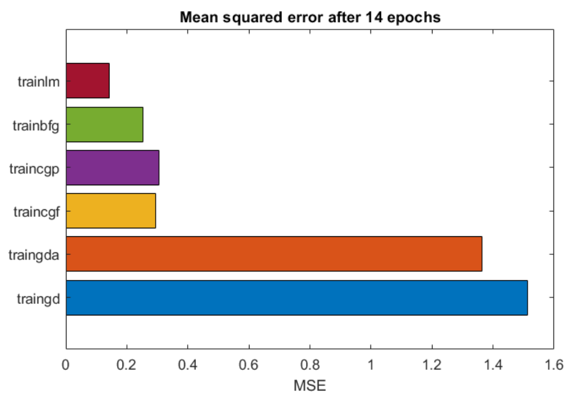
The function $f(x) = sin(x^2)$ for $x = 0 : 0.05 : 3\pi$ is to be approximated using a feedforward neural network (NN) with one hidden layer. The goal is to approximate the function using different learning algorithms and compare their performance. The learning algorithms considered are: Gradient descent (GD), Gradient descent with adaptive learning rate (GDA), Fletcher-Reeves conjugate gradient algorithm (CGF), Polak-Ribiere conjugate gradient algorithm (CGP), BFGS quasi Newton algorithm (BFG) and Levenberg-Marquardt algorithm (LM).

1.1.1 Network architecture

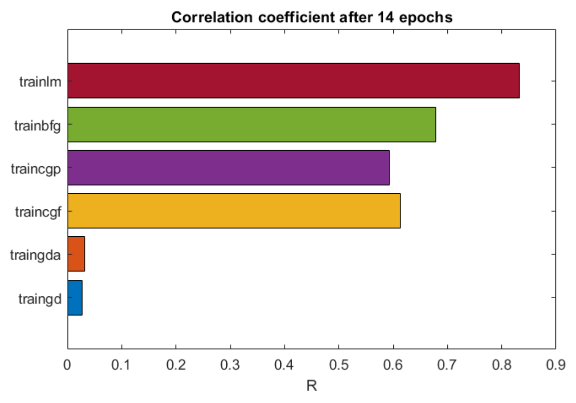
To make a fair comparison, the NN architecture is kept constant with 30 neurons in the hidden layer. The weights and biases are initiated with identical values for each learning algorithm. No validation or test sets are used and the learning algorithms are trained for the same amount of epochs. With this setup, we can see the difference in performance between the algorithms. If we chose for example 50 neurons in the hidden layer, the last four algorithms would perform very well and it would be difficult to see a difference in performance.

1.1.2 Comparison

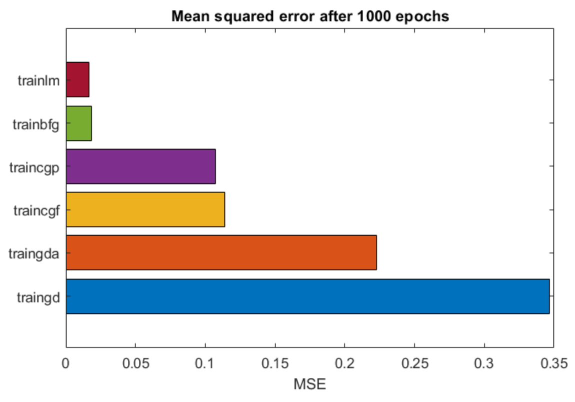
Three things will be considered when comparing the different algorithms: the time it took to train, the mean squared error (MSE) and finally the correlation coefficient R after performing regression on the output and targets. This will be measured after 14 and 1000 epochs respectively. We will let each algorithm run 100 times and then take the average value for each metric. We begin by inspecting the MSE for each algorithm, see figures 1a and 1b, we clearly see that GD is the worst performer out of all the algorithms. It gets slightly better with adaptive learning rate after 1000 epochs but is still outperformed by other algorithms. It is also clear that the LM algorithm is the best performer in both cases. BFG performs similar to LM after 1000 epochs whereas CGP and CGF are somewhere in the middle and performs similar to each other. And if we take a look at the correlation coefficients in figures 1c and 1d, it is as expected showing the same pattern as the MSE plots show.



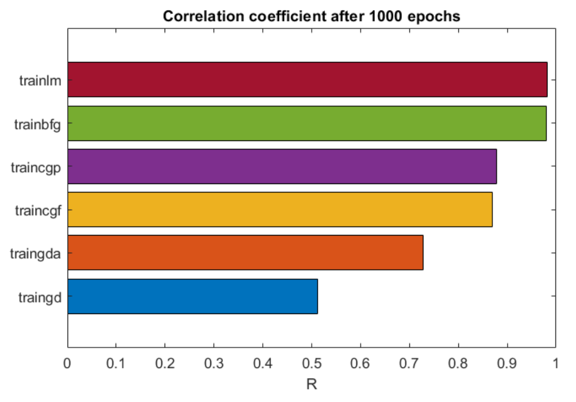
(a)



(c)



(b)



(d)

Figure 1: Plots (a),(b) show the MSE after 14 and 1000 epochs resp. for each algorithm. Plots (c),(d) show the correlation coefficient R after 14 and 1000 epochs resp. for each algorithm. Note, the x-axis values changes.

1.1.3 Noisy data

Now we want to compare the same algorithms as above but when adding noise to the data. We add Gaussian noise with $\sigma = 0.2$ to the targets and also increase the number of data. We run 1000 epochs for each algorithm with both the noisy data and the noiseless data as targets. We run this 50 times and take the average MSE and correlation coefficients R and compare them in the bar charts in figure 2. We can see that the noise does not have a very big impact on any algorithm except for GD. An issue with GD is that it converges to local minimums easily, but if we add noise to the data it will perform better.

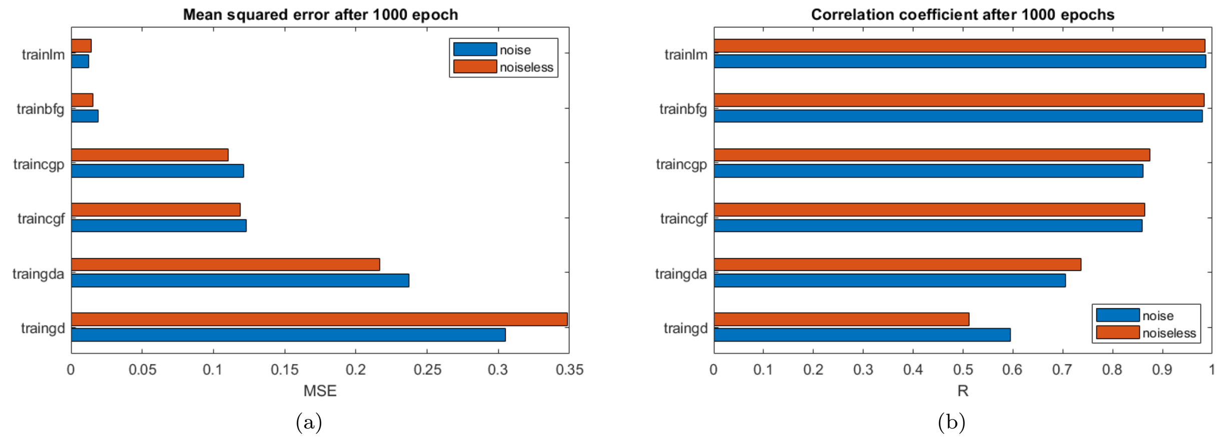


Figure 2: (a) MSE after training 1000 epochs on both the noisy and noiseless data for each algorithm. (b) correlation coefficient R after training 1000 epochs on both the noisy and noiseless data.

1.2 Personal regression

In this exercise, we use a feedforward NN to try and approximate a nonlinear function. The data set consists of 13600 data points with two inputs and one output. We take three random samples of size 1000 each and use these as training, validation and test sets when training an NN. The sets are drawn randomly and without replacements since we do not want any of the training points to occur in the test or validation set as that would mean that we would train on the same points as we test on, which would create a bias. The training set is visualized in figure 3.

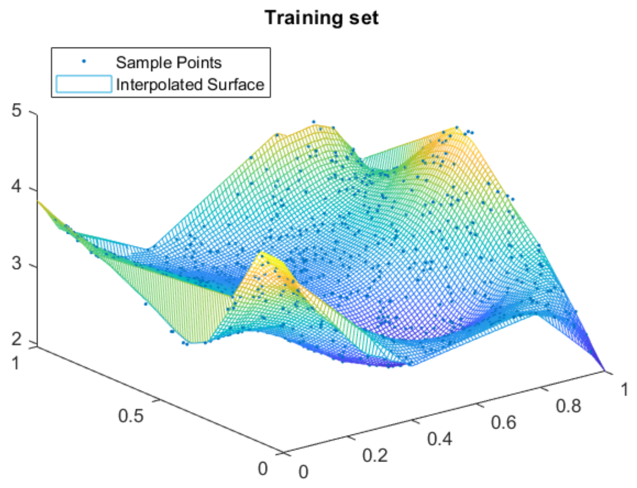


Figure 3: The training set visualized using an interpolated surface together with the data points.

1.2.1 Building and training a neural network

From the previous exercise, we saw that the LM algorithm performed well when approximating a nonlinear function and we will therefore use the same algorithm for this problem. To avoid the problem of overfitting, Bayesian regularisation will be used during training. Since we only have two features, I will use two hidden layers for this problem with the same amount of neurons in both layers and using the *tanh* activation function. To decide how many neurons were suitable, NNs with different amounts of neurons were trained for the same amount of epochs. The MSE on the test set was then computed. The error was averaged over 10 runs for each NN and the result can be seen in figure 4. The figure shows that the performance of the NN stops increasing when there is more than around 15 hidden neurons in each hidden layer. So for this problem I would use an NN with two hidden layers with 20 hidden neurons in each layer using a *tanh* activation function.

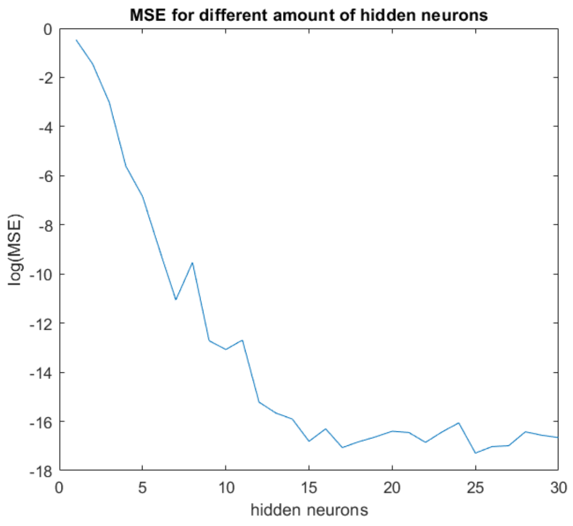


Figure 4: MSE after 500 epochs with different amount of neurons in the layers. Note the logarithmic scale.

1.2.2 Result

After training the NN for 1000 epochs, we use that network on the test set and calculate the mean squared error on the result. This was averaged over 10 iteration and came out as $MSE_{ave} \approx 5 \cdot 10^{-8}$. To visualize the result, we plot the mesh of the original test set and the network output on the test set, see figure 5. We can see in this figure that they look very similar. We plot the error level curves for the NN trained with the LM

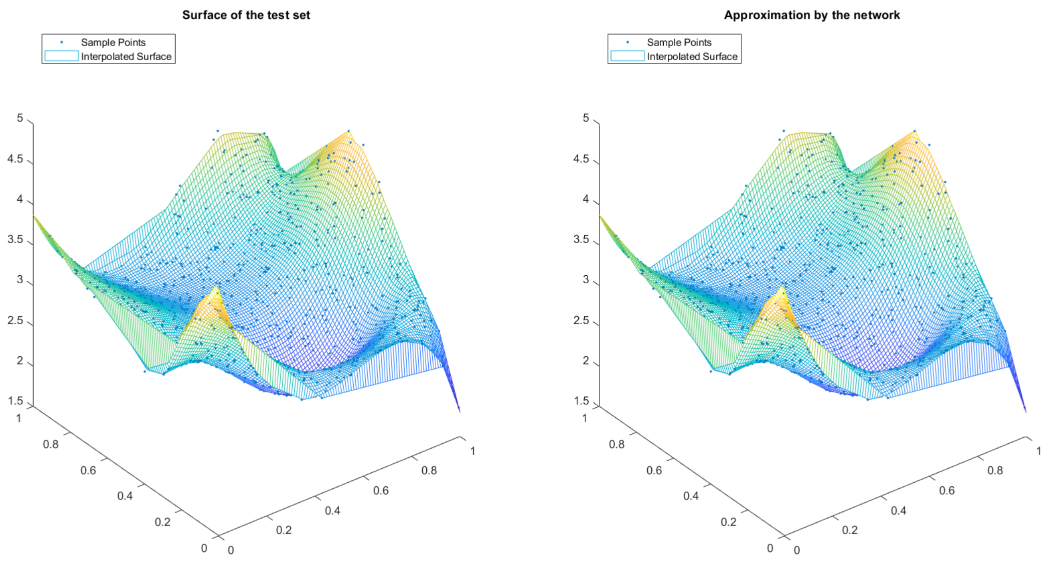
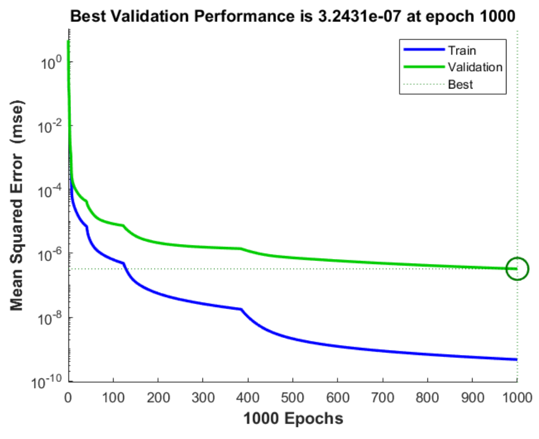
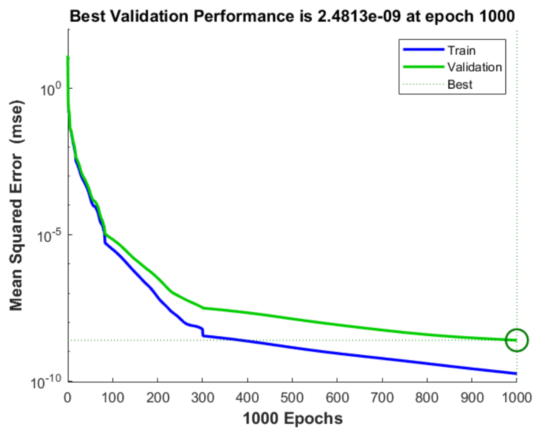


Figure 5: Interpolated surface together with data points for the test set (left) and the network output (right).

algorithm in 6 where the first plot shows the training without regularisation and the second one with Bayesian regularisation. We see that the NN performs better with regularisation and that the validation performance is closer to the training performance compared to the case with no regularisation.



(a)

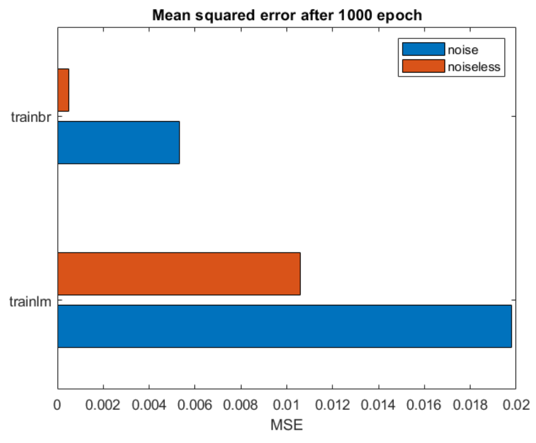


(b)

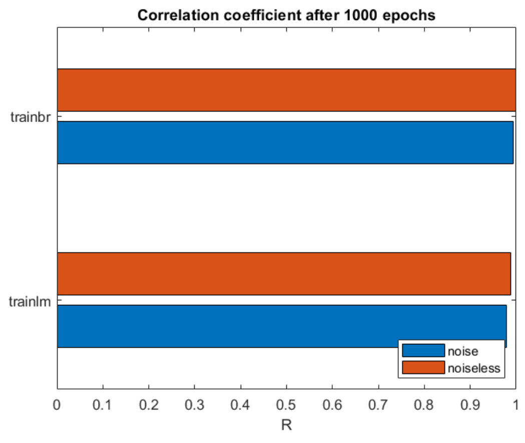
Figure 6: The error level curves when training without regularization (a) and with Bayesian regularization (b).

1.3 Bayesian Inference of Network hyperparameters

We now visit the same problem as the first exercise, but we will now use the *trainbr* algorithm in MATLAB to see if the performance changes, with the hope of reducing the problem of overfitting the NN. We will compare the algorithm with regular LM without regularisation to see if we get a better result. We do as we did under the noisy data section and compare the MSE and correlation coefficient for the two algorithms when using both the noisy and noiseless data. The result is displayed in figure 7 where we can see a clear improvement when using Bayesian regularisation. We can also see that these algorithms perform better on the noiseless data. We



(a)



(b)

Figure 7: (a) MSE and (b) correlation coefficient R; after training 1000 epochs on both noisy and noiseless data with and without regularization.

would now like to see how the performance compare when using an overparametrized NN with many neurons. Theoretically we should see the same pattern as before that the algorithm with regularisation outperforms the one without. For this we used 300 hidden neurons on the noiseless data and with $x = 0 : 0.05 : 3\pi$ and a maximum of 50 epochs. The MSE averaged over 10 runs is show in table 1. We can see that the performance for the LM algorithm is much better with 300 neurons compared to 30 neurons from the first exercise. However, with regularization the MSE is much smaller so we can confirm that regularization is very important when dealing with an overparametrized NN to reduce overfitting.

Algorithm	MSE
trainlm	$2.8915 \cdot 10^{-8}$
trainbr	$4.2581 \cdot 10^{-19}$

Table 1: Average MSE using an overparametrized network of 300 neurons with and without regularization.

2 Exercise Session 2: Recurrent neural networks

2.1 Hopfield networks

A Hopfield network is a recurrent neural network (RNN) with one layer of neurons and is fully interconnected. It is initialized with stored patterns that acts as the networks' attractors. The network then takes an input and updates in a synchronous way until it reaches one of the attractors. So after receiving an input, the network should evolve to the closest attractor.

2.1.1 Hopfield network with two neurons

We initialize a Hopfield network with the three attractors

$$T = \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & -1 \end{bmatrix}^T.$$

To test the dynamics of the network, some points were given as input for the network and then the evolution of the network was examined. To test what happens when an input was exactly in the middle of two points, a symmetric grid was used to choose the input points. We let the network synchronously update 100 iterations for each input and the evolution of each input was recorded, see figure 8 for the trajectories for each input. By doing this, we saw that the network had another set of attractors apart from the initialized attractors, these are the green circles in the figure and they are

$$T = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 & 1 & 0 \end{bmatrix}^T.$$

We see that the point [-1 1] is an inverse of one of the initial attractors, so this is a local minimum for the network. The rest are points exactly in the middle of the attractors. We get these spurious states since they are local minimas of the network so if the network gets these points as an input, it will not leave these states. The amount of iterations it took before the network converged was also recorded for each input and was approximately 6.4 iterations on average to converge to one of the initial attractors or a spurious state.

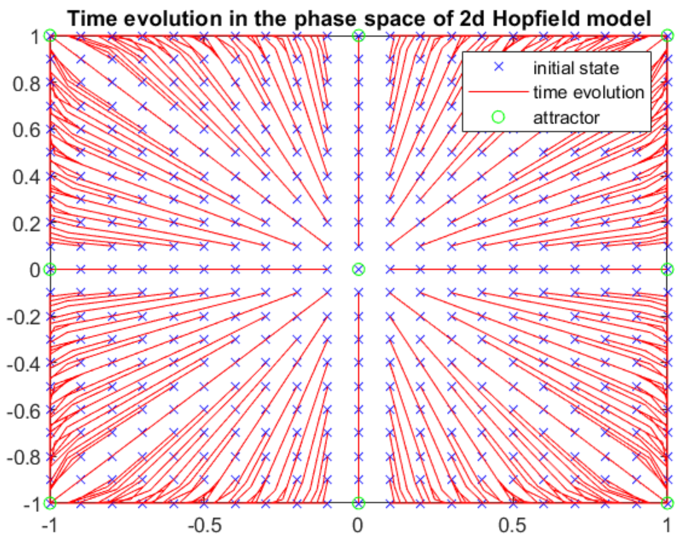


Figure 8: Evolution of the network for each input shown as trajectories. Final point of the evolution is marked.

2.1.2 Hopfield network with three neurons

The same process as in the previous section was executed on a Hopfield network with three neurons and the initial attractors

$$T = \begin{bmatrix} 1 & 1 & 1 \\ -1 & -1 & 1 \\ 1 & -1 & -1 \end{bmatrix}^T.$$

We now use a three dimensional grid to once again see what happens when the input to the network is in the middle of attractors. We notice that it takes longer for the network to converge to an attractor than in the case with two neurons, it now takes approximately 55.5 iterations on average to converge to an attractor. In this is case however, we always converge to one of the initial attractors so we have no unwanted attractor. The evolution of the network for this case can be seen in figure 9, note that the figure only shows a few inputs from what was tested.

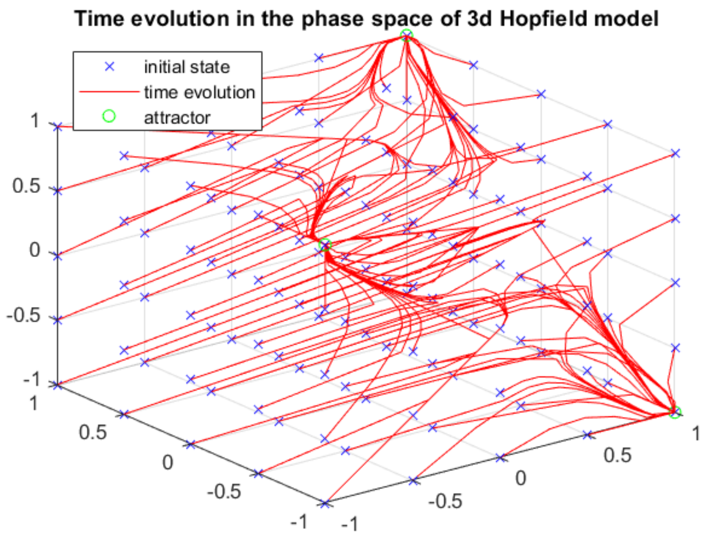


Figure 9: Evolution of the network for each input shown as trajectories. Final point of the evolution is marked.

2.1.3 Applying a Hopfield network on handwritten digits

A Hopfield network is initialized with the digits 0-9 as attractors. To test its ability to correctly reconstruct a digit, a distorted version of a digit is fed as input to the network. When the noise is small, the network correctly retrieves the correct pattern after only a few iterations. However when the noise increases, the amount of needed iterations to converge increases as well. In addition, if too much noise is added the network can converge to the wrong digit. This wrong reconstruction happens most frequently for the similar looking digits, for example the digits 4 and 1, however it seems to happen for most digits if the noise level is really high. It is expected that the network has a hard time when the distortion is really high since even for a human eye it would be hard to predict which digit it is. In conclusion, we can say that a Hopfield network effectively returns the correct digit when given a slightly distorted digit as input. When given a highly distorted digit, it requires more iterations and has a chance of retrieving the wrong digit. A few examples of reconstructed digits are shown in figure 10.

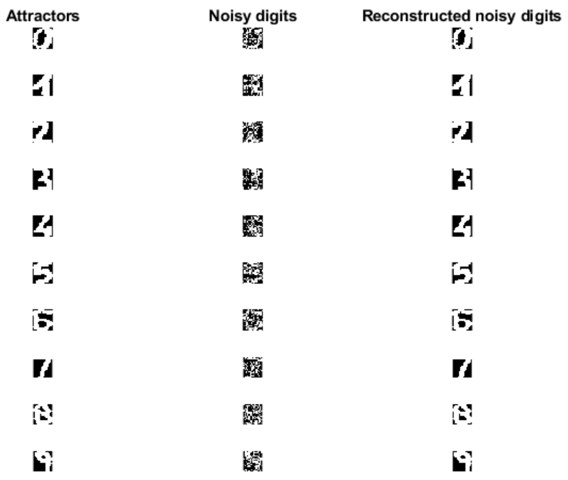


Figure 10: Reconstruction of distorted digits by a Hopfield network after 10 iterations

2.2 Time series prediction with a multilayer perceptron

The goal of this exercise is to create and train an NN on a time series data set and make predictions about the behaviour in the future. The data set considered was the Santa Fe data set containing the behaviour of a chaotic laser, described as a nonlinear dynamical system. The data set contains 1000 given data points that acts as the training set and then an additional 100 data points that is the test set we want to predict as accurately as possible. The Santa Fe data set can be seen in figure 11 with the division between training and test sets.

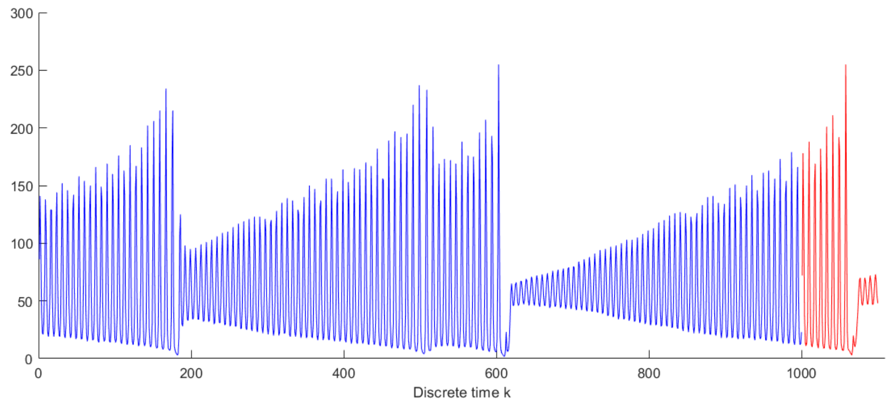


Figure 11: The Santa Fe data set where the training set is blue and the test set is shown in red.

We started off by standardizing the data set and then creating a multilayer perceptron (MLP) with one hidden layer. The training of the MLP is done in feedforward mode as

$$\hat{y_k} = w^T tanh(V[y_{k-1}; y_{k-2}; ...; y_{k-p}] + \beta), \tag{1}$$

where p is the lag, describing how many data points will be considered to predict the next point. And then to make the network predict the next data point we use the previously predicted values according to

$$\hat{y_k} = w^T tanh(V[\hat{y}_{k-1}; \hat{y}_{k-2}; ...; \hat{y}_{k-p}] + \beta). \tag{2}$$

The training of the MLP was done using the LM algorithm. The amount of hidden nodes was initially fixed to 50, and to decide the lag p, we trained and simulated a network for p in the range of $[1, 100]$ and calculated the root mean square error (RMSE) on the test set. We did this three times for each value of p and the RMSE was then averaged. This to get an idea of a good value for p. We see in figure 12 that the RMSE is not very stable but the value for p should be at least 20 to get a good result. It seems that the network performs slightly better with a higher value of p, but it is irregular and demands more computational power p is increased.

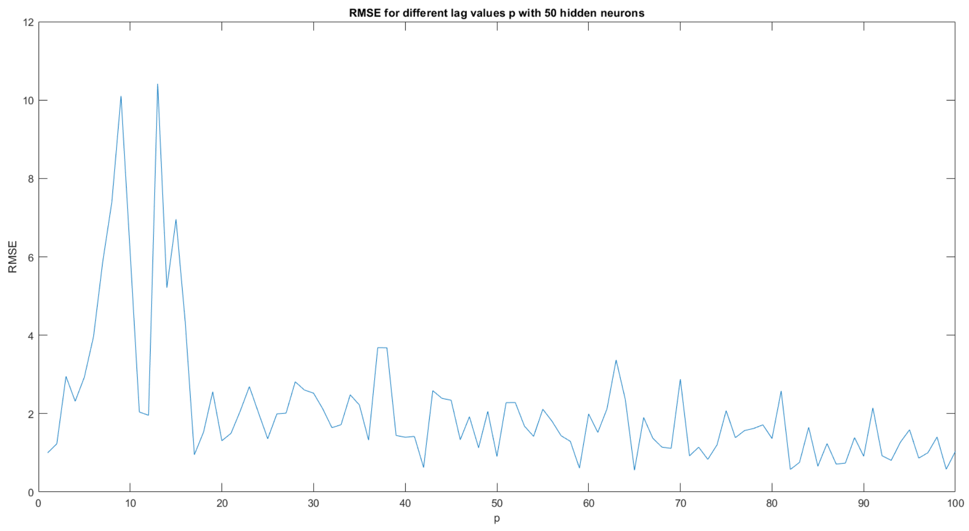


Figure 12: The root mean square error on the test set using different values for the lag, p.

2.2.1 Result

With an MLP with one hidden layer of 50 hidden neurons and with input lag $p = 50$ we get the result shown in figure 13. The results you get after training an MLP with these parameters differs quite a bit, usually it performs very well until time step 60 and after that it has a harder time predicting the behaviour of the laser. This iteration performed well if you compare it to the average networks performance.

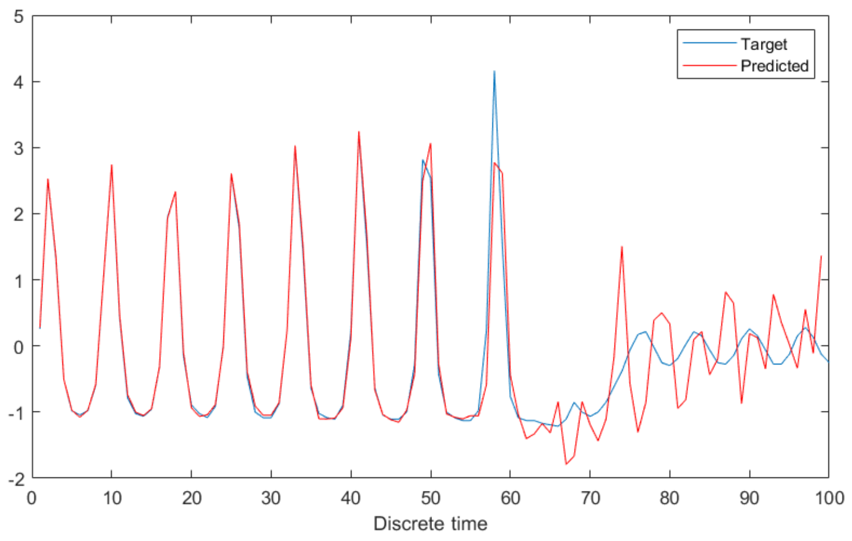


Figure 13: The target values with the predicted values from the MLP with 50 hidden neurons and lag $p = 50$.

2.2.2 Time series prediction with a Long short-term memory network (LSTM)

We are now interested to predict the same time series as in the previous section but this time with an LSTM. The network architecture consists of a input layer, an LSTM layer with 200 neurons, a fully connected layer and finally a regression layer. The network was then trained for 300 epochs with the 'Adam' optimization algorithm with an initial learning rate of 0.005 that gets reduced every 100 epochs by a factor of 0.2. Increasing the input lag now does not make the learning much slower as it did with the MLP. When using no input lag for training and predicting the following 100 time steps we saw that it performed better than the MLP. If we add input lag, the performance increases but it is not able to perfectly predict the following time steps. When using $p = 30$, the LSTM predicts a good result, even better than what the MLP produced in its best case. See the result in figure 14. We conclude that the LSTM trains faster than the MLP, it produces a better result without any input lag and finally it predicts the following 100 time steps well when input lag is used.

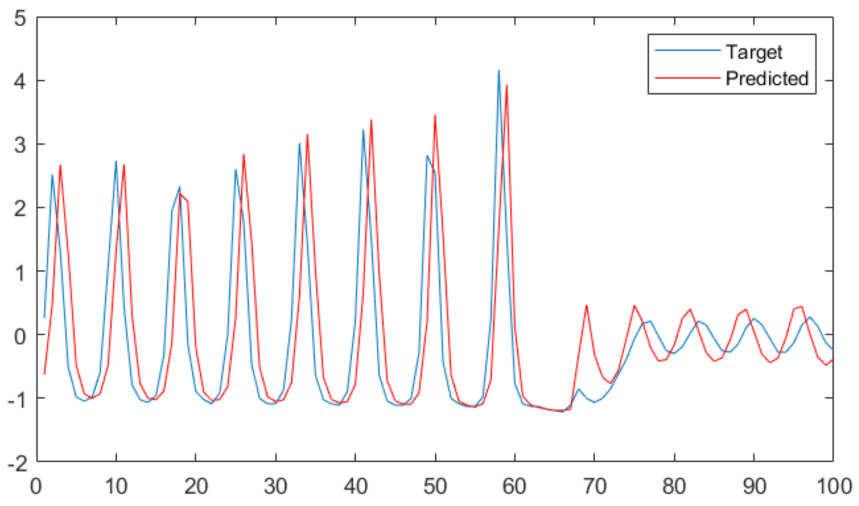


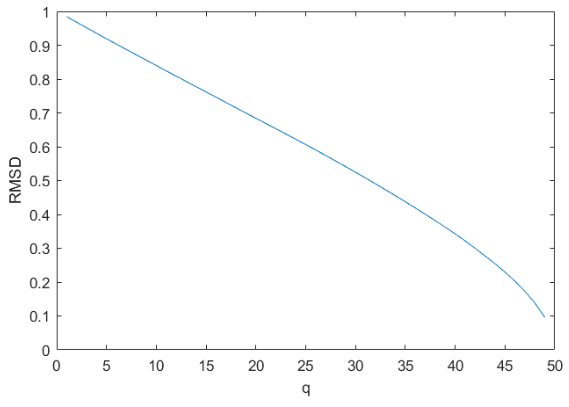
Figure 14: The target values together with the predicted values from the LSTM with input lag $p = 30$.

3 Exercise Session 3: Deep feature learning

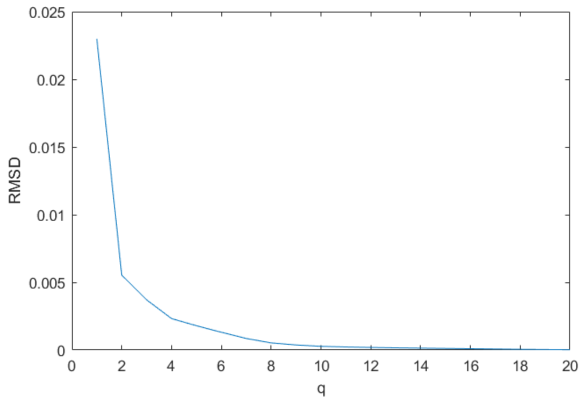
3.1 PCA on random and highly correlated data

A data set consisting of 500 data points with 50 dimensions is randomly generated from a Gaussian distribution. A dimensionality reduction is performed by applying principle component analysis (PCA) on the data set. The dimension is reduced to q where q ranges from $[1, 49]$, the data set is then reconstructed using these q dimensions. We finally compare how well the reconstruction is by computing the root mean squared difference (RMSD) between the original data set and the reconstructed data set. In figure 15a we see that the RMSD linearly decreases when q increases.

We now consider the data set **choles_all** that consists of 264 highly correlated data points with 21 dimensions. We perform the same analysis on this data set as before and in the figure 15b we see a very different result from before. The reason is that when we now have correlated data, the largest eigenvalues will be a big percentage of the sum of all eigenvalues and therefore even a few components will contain a lot of information about the data. So when using a correlated data set, we can greatly reduce the dimensionality without losing much information about the actual data set, this comes very handy when dealing with big data sets with many dimensions.



(a)



(b)

Figure 15: RMSD between the original data set and the reconstructed data set using PCA with q dimensions. (a) is the random data set and (b) is the highly correlated data set.

3.1.1 PCA on handwritten digits

We are now going to do a similar task as in the previous section on a more concrete problem, namely apply PCA on handwritten digits. We use the digit 3 from the US Postal Service database, there we have 500 images of size 16x16 pixels. First of we plot the mean 3, which we get by taking the average of all image, this is displayed in figure 16.

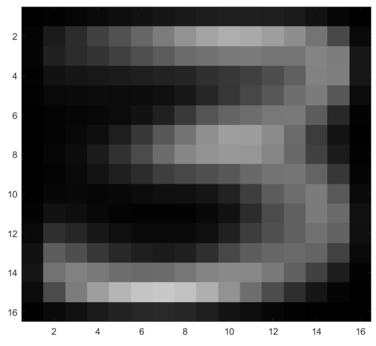


Figure 16: The mean image from the data set.

Next step we do is calculate the covariance matrix of the data set and then from this computing the eigenvalues and eigenvectors. The eigenvalues are plotted in figure 18a from largest to smallest, here we see that the eigenvalues falls rapidly in magnitude and after the 50 largest eigenvalues they are close to zero. We compress the data by projecting it onto one, two, three and four principal components and then we plot the reconstructions for some of the images, see figure 17.

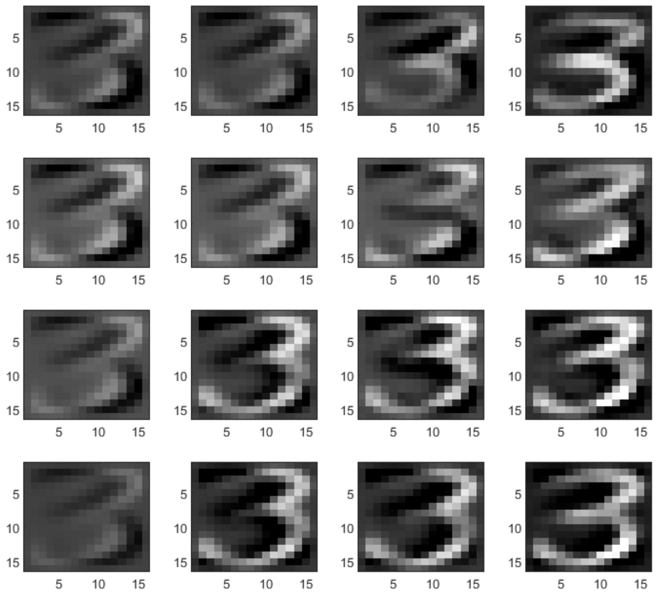
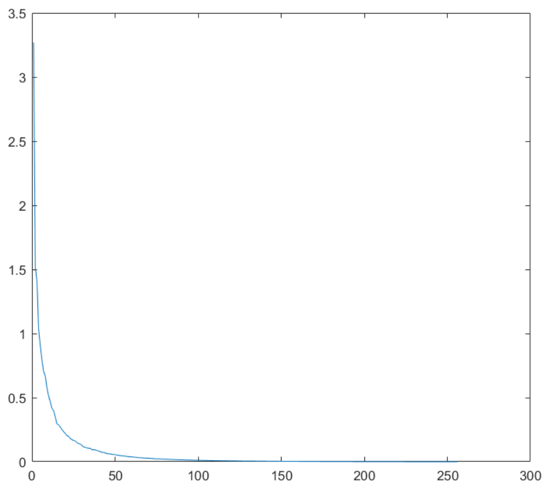
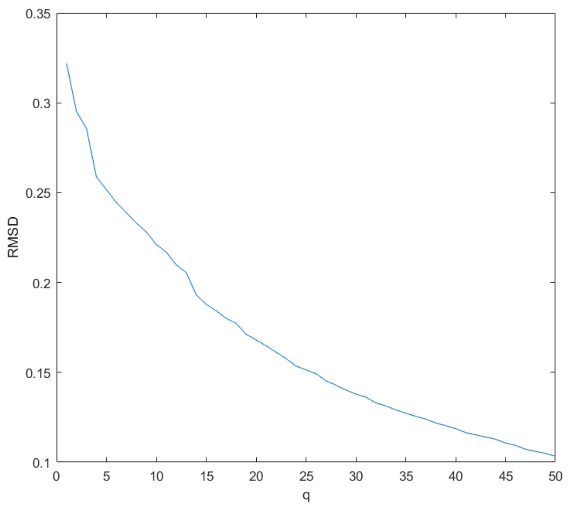


Figure 17: The reconstruction using 1, 2, 3 and 4 principal components (from left to right), each row is a different original image.

Now the reconstruction is done for q in the range of $[1, 50]$ and the RMSD between the reconstructed images and the original images is computed. The RMSD is plotted against the number of principle components q in figure 18b. Now if we were to choose $q = 256$, that would imply no dimensionality reduction and we would expect to retrieve the original data after reconstruction and therefore no reconstruction error. This is also what we get in reality.



(a)



(b)

Figure 18: (a) the values of all eigenvalues of the covariance matrix sorted from largest to smallest. (b) RMSD between the original data set and the reconstructed data set using PCA with q dimensions.

3.2 Stacked Autoencoder

We are interested in comparing the performance of a stacked autoencoder to a normal multilayer neural network for handwritten digit classification. We begin with a default stacked autoencoder consisting of two autoencoders and a softmax layer, the first and second autoencoder has 100 and 50 neurons respectievly. The first autoencoder and the softmax layer is trained for a maximum of 400 epochs, and the second autoencoder for 100 epochs. This stacked autoencoder has an accuracy of 83.7% without fine tuning and we will use this as our reference point. By changing the max epochs, number of autoencoders and number of neurons in autoencoder, we will try to improve on this accuracy. First of all, we notice that the training of the first autoencoder takes a long time and that its performance barely increases after 200 epochs, so we change this to save some time. Another thing we notice is that the performance during training for the second autoencoder is still improving quite rapidly after 100 epochs so we change the max epochs to 200 for this one also. By only changing the max epochs for both autoencoders to 200 we raised the accuracy to approximately 95% without fine tuning. When adding an additional autoencoder to the stacked autoencoder, either one before with 200 neurons or one after with 25 neurons decreases the accuracy heavily. Since we can reach 95% with only two autoencoders, we regard this as a good number of autoencoders for this problem. The final alteration we do is the amount of neurons in each autoencoder. First off, we halved the amount in both only to see that the accuracy got worse. Then the neurons were doubled and the accuracy jumped to almost 99%, The accuracy improved further to 99.1% with 150 neurons in the second autoencoder. So the architecture that got us the best accuracy without fine tuning was the following: training two autoencoders for 200 epochs with 200 and 150 hidden neurons respectively. This outperforms the multilayered neural network with the same amount of layers and hidden neurons by 3%.

Now if we also finetune the stacked autoencoder, the accuracy improves even more. For example, with the chosen architecture as before we bump the accuracy up to 99.6%. The finetuning is done by performing back-propagation on the whole multilayer network after the autoencoders and the softmax layer are stacked. It is retrained in a supervised fashion using the training targets. The objective of finetuning is to adjust the weights of the stacked autoencoder to improve the accuracy by using the whole training set.

3.3 Convolutional neural network

We take a look at a pre-trained convolutional neural network (CNN) to extract features from images of different types of flowers. The pre-trained model to be used is the AlexNet that consists of 23 layers where five of these are convolutional layers directly followed by a ReLu layer. AlexNet takes a color image of size 227 by 227 pixels as input and classifies that image as one of a 1000 classes. It is in the convolutional layers that the main feature extraction happens, if we start by taking a look at the first of these layers we see that the dimension of the weights are 11x11x3x96. This means that there are 96 convolutional kernels of size 11x11x3, a visualization of these convolutional kernels learned by the network is visualized in figure 19. There we can see the 96 features that are extracted in the first convolutional layer and what they look like.

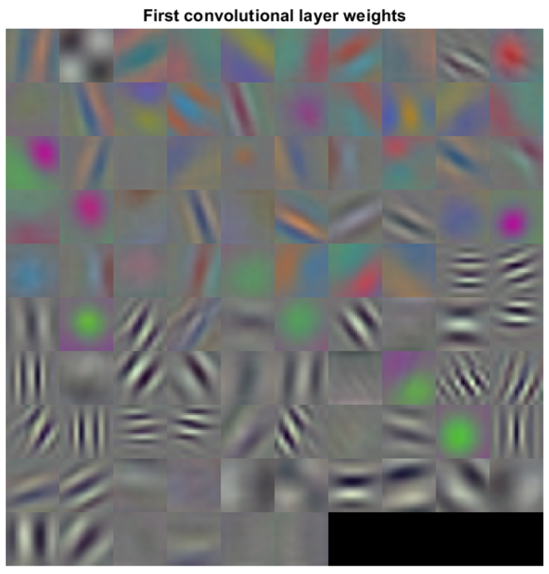


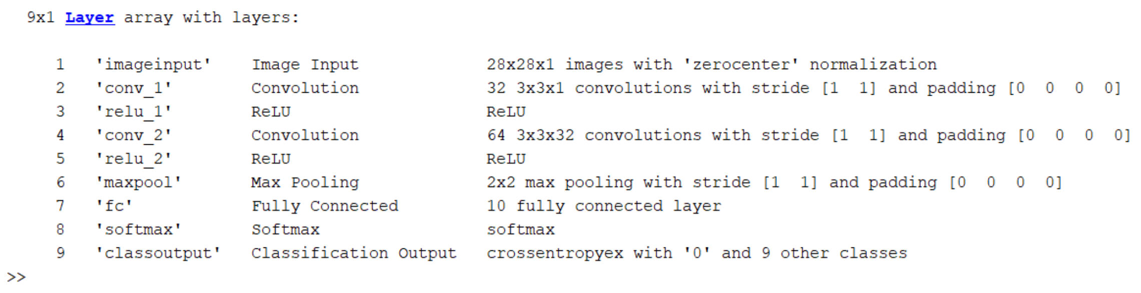
Figure 19: The weights of the first convolutional layer, 96 kernels of size 11x11x3.

By looking at the first five layers of the network, we want to know the input dimensions to the sixth layer. First of all we know that the input to the second layer is by the dimension 227x227x3 and here we have a convolutional layer with 96 kernels of dimensions 11x11x3 and a stride of 4. The formula for the output dimensions from a convolutional layer and a max-pooling layer with kernel size K, stride S and input volume W is $\frac{W-K}{S}+1$. So we get the output dimension from the second layer as 55x55x96 since we have 96 kernels. The two following layers are a ReLu and a Cross Channel Normalization layer, which do not change the dimensions. Then we have a max-pooling layer with kernel size 3 and a stride of 2, so by using the same formula as before we know that the dimensions after this layer is 27x27x96 and this is also the input to the sixth layer.

We do the same calculations for the remaining convolutional and pooling layers until we reach the first fully connected layer. We do this to see how the dimensionality have changed from the input. The same formula as before was used on the remaining layers and up until the first fully connected layer, the dimension have reduced to 6x6x256. This means that we have a total of 9216 neurons as input before the classification part of the network, this is only a fraction of the 154587 neurons in the input layer. A big advantage of using a CNN over a fully connected network in image classification is that a CNN uses local connectivity. The fully connected network instead receives information from all parts of the image and treats pixels far away the same way as pixels which are close together. Another advantage with a CNN is that the training is easier and it requires fewer parameters than a fully connected network.

3.3.1 Small CNN on handwritten digits

We test out a CNN of a smaller size to classify handwritten digits. The images of the digits are of size 28 by 28 pixels and the CNN consists of a convolutional layer with 20 kernels and kernel size 5, followed by a ReLu and a max-pooling layer and finally a fully connected soft-max layer and a classification layer. This architecture has an accuracy of 93%. In an attempt to improve this, we added an additional convolutional and ReLu layer between the first ReLu layer and the max-pooling layer. Also the amount of filters in the convolutional layers were 32 and 64 respectievly with a kernel size of 3, see figure 20 for the network architecture. By only applying these changes, we got an accuracy of 99%. Having convolutional layers right after each other is powerful as seen in the AlexNet in the previous section, so this explains the improvement of the CNN after these small changes.



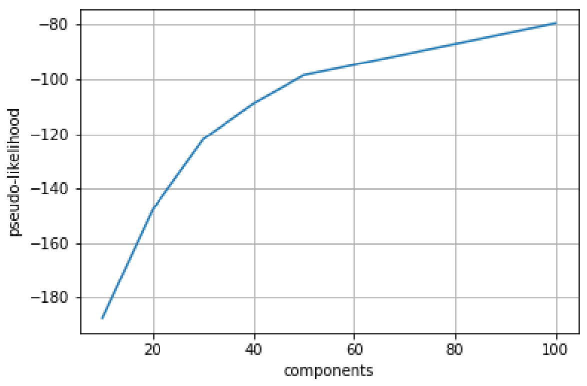
```
  9x1 Layer array with layers:

     1   'imageinput'    Image Input             28x28x1 images with 'zerocenter' normalization
     2   'conv_1'        Convolution             32 3x3x1 convolutions with stride [1  1] and padding [0  0  0  0]
     3   'relu_1'        ReLU                    ReLU
     4   'conv_2'        Convolution             64 3x3x32 convolutions with stride [1  1] and padding [0  0  0  0]
     5   'relu_2'        ReLU                    ReLU
     6   'maxpool'       Max Pooling             2x2 max pooling with stride [1  1] and padding [0  0  0  0]
     7   'fc'            Fully Connected         10 fully connected layer
     8   'softmax'       Softmax                 softmax
     9   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes
>>
```

Figure 20: The CNN architecture that yielded the best result on classifying handwritten digits.

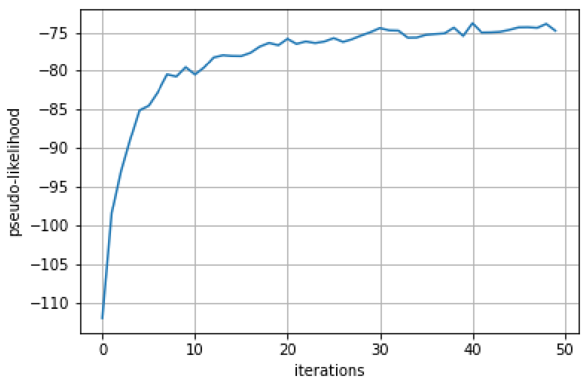
4 Exercise Session 4: Generative models

4.1 Restricted Boltzmann machines

A restricted Boltzmann machine (RBM) is trained on the MNIST dataset of handwritten digits. The trained RBM can be used to reconstruct unseen data to evaluate the performance. We use this to investigate what effect the training parameters **epochs** and **components** have on the RBM. When choosing these hyperparameters we use the metric *pseudo-likelihood* during training. Firstly, to decide the optimal amount of components, we train an RBM with different amount of components for the same amount of iterations and the same learning rate. This is done to see how the pseudo-likelihood changes for different amount of components and we then pick the parameter value for when the likelihood is the highest, see figure 21a for the result. Since the RBM takes a long time to train, we limited the amount of components to 100. We see in the figure that the more components we have, the higher the likelihood is. So we choose an RBM with 100 hidden units. To decide the optimal amount of iterations, we ran the training with 100 components for 50 iterations and saw how the pseudo-likelihood changed over time, see figure 21b. The amount of iterations doesn't increase the likelihood notably after about 20-25 iterations, so we will use this amount.

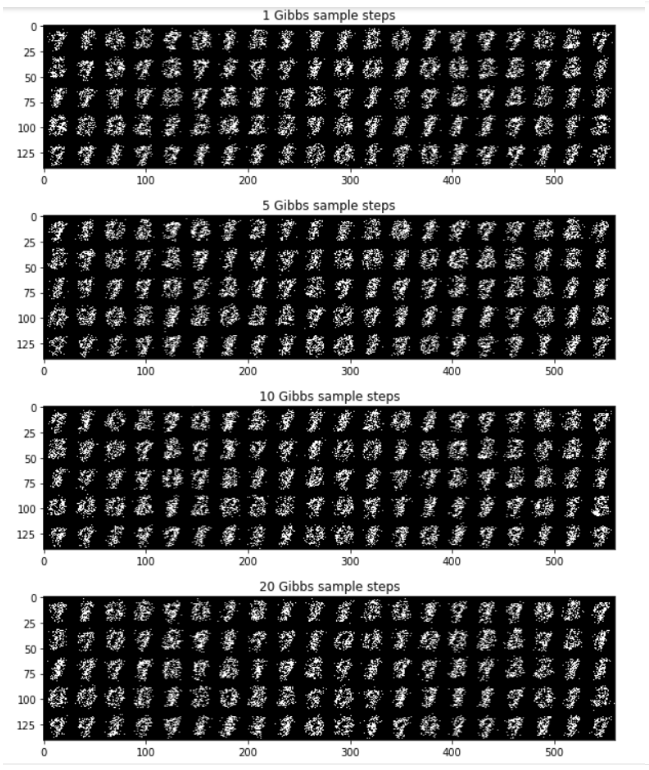


(a)

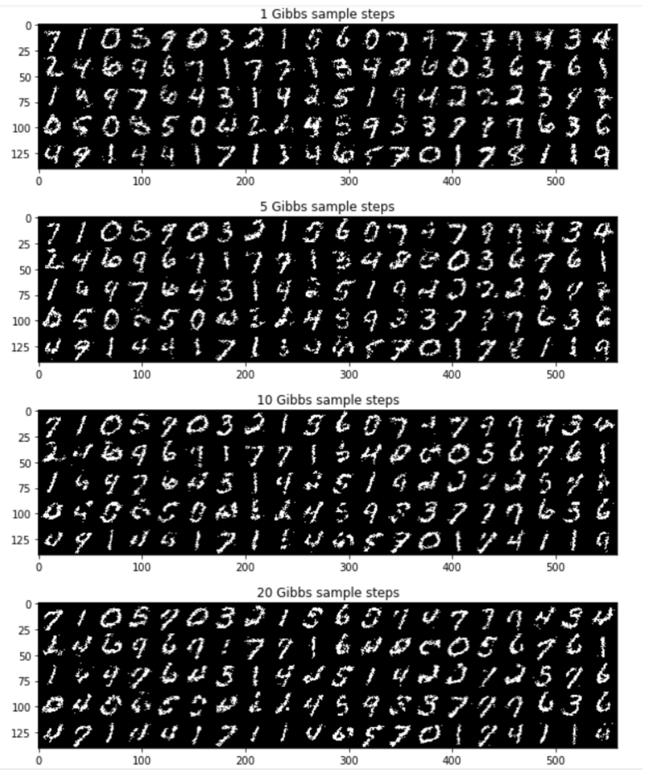


(b)

Figure 21: (a) The pseudo-likelihood after training 20 epochs for different amount of components. (b) The pseudo-likelihood per iteration using 100 components.



(a)



(b)

Figure 22: The reconstruction of images using different amount of Gibbs sample steps, (a) is an RBM trained with 10 components and (b) with 100 components.

Now to show the difference it makes to change these parameter we show the reconstructed images in figures 22a and 22b from an RBM with 10 and 100 components respectively and with different number of Gibbs sampling steps. The original images are shown in figure 23. It clearly does a much better job at reconstructing images when using 100 hidden units than with 10. We can also conclude that 1 or 5 Gibbs steps performs the best in the case with 100 hidden units. Now if we give the RBM an image with some rows removed and try to reconstruct the full image we see that this is difficult if too many rows have been removed. Furthermore, a better trained Boltzmann machine performed much better on this task. But even when using the RBM that produced the reconstructed images in figure 22b we got a rather poor result when removing more than 10 rows. The worst result was retrieved when rows in the middle of the images were removed.

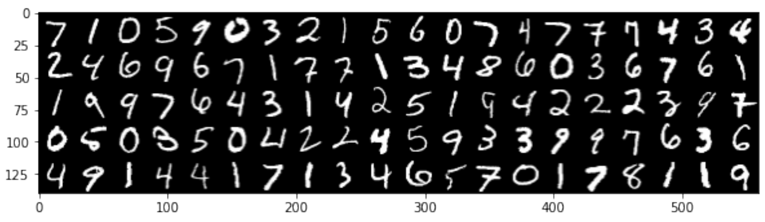
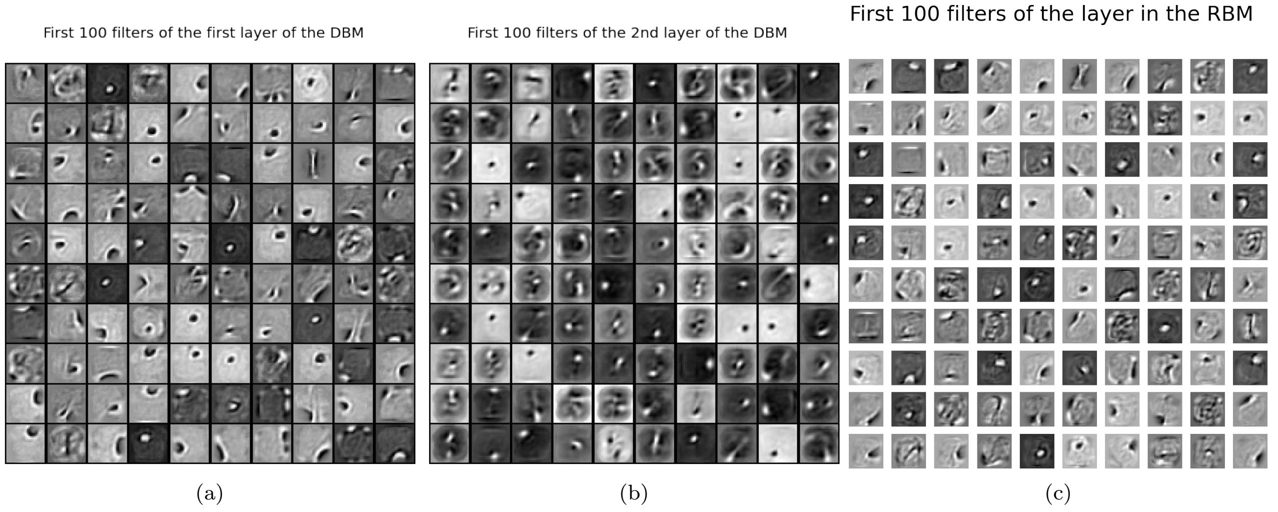


Figure 23: The original images that were reconstructed in figure 22.

4.2 Deep Boltzmann Machines

A deep boltzmann machine (DBM) can be seen as a series of RBMs stacked on top of each other. We use a DBM consisting of two stacked RBMs that has been pre-trained on the MNIST database and we would like to compare this with the RBM in the previous section. If we start by looking at the filters extracted from both layers in the DBM in figures 28a and 28b we see that they capture different features of the digits. The first layer captures mostly simple shapes such as circles and the positions of these. The second layer seem to capture more complex and connected shapes, there it is almost possible to see what the underlying digit is. If we compare this to the filters of the only layer in the RBM from the previous section in figure 24c, we see that it is similar to the first layer of the DBM where is extracts mostly basic features. Now if we take a look at the reconstructed digit from the DBM in figure 25 we can see that the digit now looks very similar to the original and better than what the RBM produced in figure 22b.


First 100 filters of the first layer of the DBM
First 100 filters of the 2nd layer of the DBM
First 100 filters of the layer in the RBM
(a)
(b)
(c)
Figure 24: The first 100 filters of the first and second layers in the DBM (a), (b) and the layer in the RBM (c).

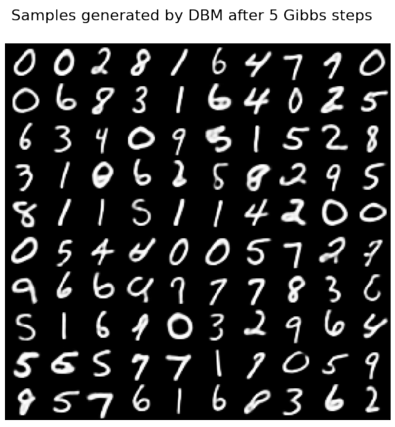


Figure 25: The reconstructed digits by the DBM using 5 Gibbs sample steps.

4.3 Generative Adversarial Networks

We train a Deep convolutional generative adversarial network (DCGAN) on images on dogs from the CIFAR data set. There is no objective way to evaluate the performance of a GAN model. What we can do is monitor the accuracy and loss for both the discriminator and the generator, see figure 26. Also a subjective inspection of the produced images can be done to decide if the images look ”real” enough. Since the training is very unstable as we see in figure 26, it is no guarantee that the performance will increase each iteration. Therefore a subjective examination of the produced images to decide if the training should stop is a valid method.

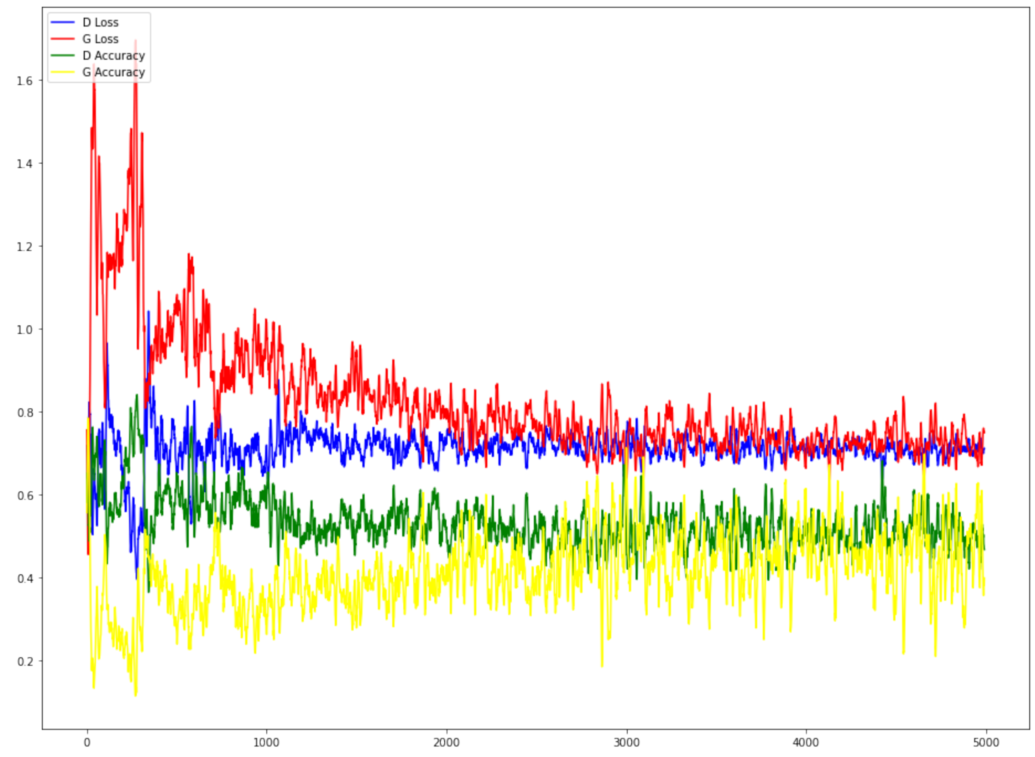


Figure 26: The loss and accuracy of the generator and the discriminator during the training of the DCGAN.

4.4 Optimal transport

4.4.1 Optimal color swapping

Optimal transport will be used for transferring color between two images using their color histograms. The two images that we used can be seen in the first column of figure 27. The resulting images after color transferring are shown in the two other columns of figure 27. The middle column uses the Earths’ mover distance transport and the last column uses Sinkhorn distance transport. In the images we can see that it is different from simply just swapping the pixels.

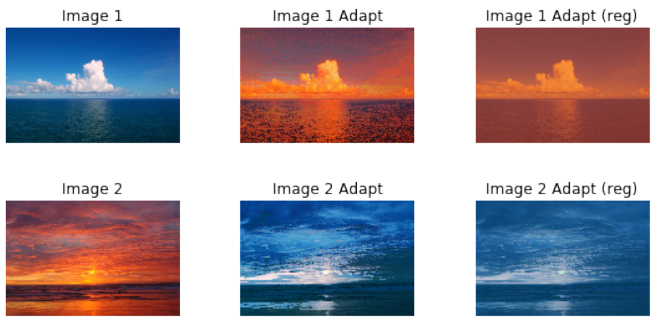
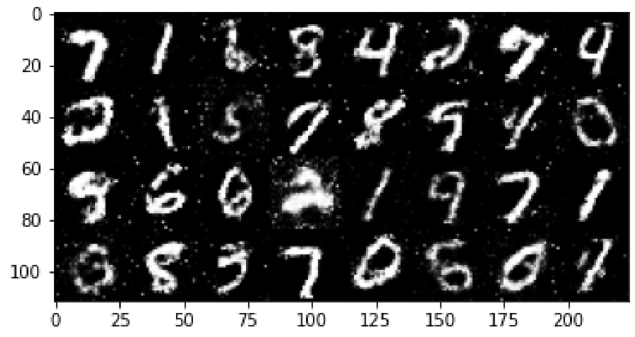


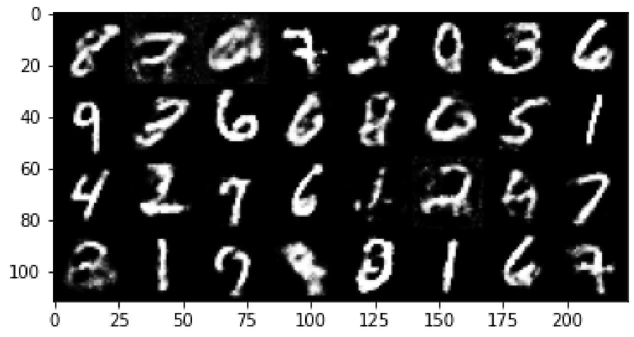
Figure 27: The original images (left) and the resulting images after color transferring (middle and right).

4.4.2 Fully connected minimax GAN vs. Wasserstein GAN

We want to compare the performance between a fully connected minimax GAN and a Wasserstein GAN. First of all, we can see by looking at the generated images that the performance of the standard GAN between iterations is quite unstable. The quality of the images are also not as good compared to what the Wasserstein GAN produces see figure 28. Also, the training of the Wasserstein GAN is more stable over the iterations, however the training is a lot slower.



(a)



(b)

Figure 28: The reconstructed images using a standard GAN (right) and a Wasserstein GAN (right).