1 Function approximation and learning from noisy data

1.1 Function approximation

Starting from the given code, a neural network is trained with different algorithms, consisting of one hidden layer with 50 neurons. First of all, the gradient descent training algorithm is compared to the Levenberg-Marquardt (LM) training algorithm. The performance of the neural net is shown in Figure 1. It can be seen that the gradient descent algorithm is outperformed by the LM algorithm. The gradient descent algorithm does not return a good function estimation, even after 1000 epochs of training (R=0.602). The algorithm probably converged to a local minimum for the set of weights of the network. Because the learning rate is fixed during training, this value will not be optimal during the whole training. The GD with adaptive learning rate addresses this problem. On the other hand, the LM algorithm converges faster than the gradient descent algorithm. Moreover, the weights that are obtained are optimal, as the neural net already does a perfect function estimation after 15 epochs (R=1).

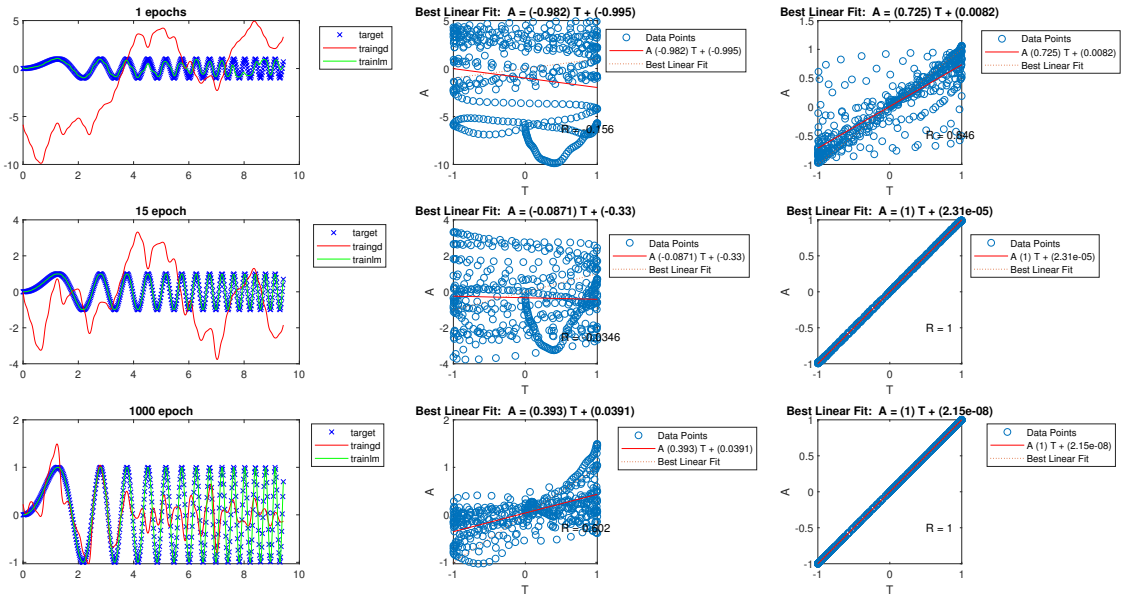
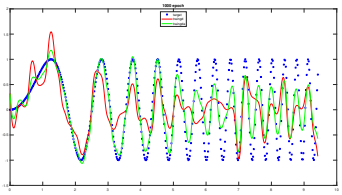
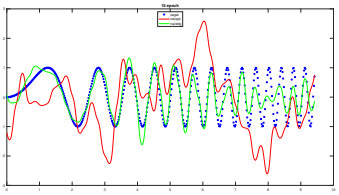


Figure 1: function estimation with neural nets: gradient descent (red) vs. LM (green) training algorithm after 1, 15 and 1000 epochs

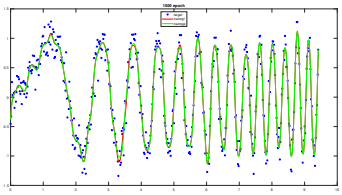
With all algorithms, the performance improves as the network is trained for more epochs. The gradient descent algorithm has the lowest performance of all algorithms. Gradient descent with adaptive learning rate performs better, but still has a higher average error than other algorithms. The quasi-Newton (BFG) algorithm is better and gives already a good function estimation after 15 epochs (R=0.91). The performance of the Fletcher-Reeves (FR) and Polak-Ribiere algorithms are similar with an average error of $\pm 0.01\%$ after 1000 epochs. Nevertheless, the LM algorithm performs the best for this problem, because it quickly converges to an optimal set of weights. While the other algorithms need more iterations to train good weights, or converge quickly to a less optimal set of weights. The results are shown in Figure 2 on the following page.



(a) GD (red) vs GDA (green) after 1000 training epochs



(b) GD (red) vs. BFG (green) after 15 training epochs



(c) FR (red) vs PR (green) after 1000 training epochs

Figure 2: Comparison function approximation of the different training algorithms (blue=target)

1.2 Learning from noisy data: generalization

In this section, noise is added to the target function. The same MLP architecture with one hidden layer with 50 neurons is trained with the different algorithms. A comparison of the average MSE across networks trained with the different algorithms is shown in Table 1. With noise, the MSE is mostly higher than without noise. Also, the function approximation is never perfect after 1000 epochs for all training algorithms (it cannot be because the noise is random).

	sin function no noise			sin function with noise ($\sigma = 0.2$)		
MSE	1 epoch	15 epochs	100 epochs	1 epoch	15 epochs	100 epochs
Gradient descent	8.56	2.77	0.28	15.45	2.93	0.29
GD with adaptive learning	4.75	1.47	0.10	15.45	2.00	0.13
Quasi-Newton	1.63	0.080	1.8E-4	8.23	0.16	0.031
Fletcher-Reeves	1.63	0.23	8.9E-4	8.23	0.50	0.034
Polak-Ribiere	2.75	0.25	8.2E-4	4.31	0.43	0.032
Levenberg-Marquardt	0.13	7.8E-5	6.5E-8	0.12	0.031	0.026

Table 1: comparison MSE in function of epochs of the different training algorithms for a sin function with and without noise.

The difference in performance of the algorithms is similar as with the noiseless case. The Levenberg-Marquardt algorithm is also the best option for regression on functions with noise. It yields a low MSE after already 15 epochs of training. The Quasi-newton algorithms comes in second best. The different algorithms all train the neural net in a few seconds. Training for 100 epochs takes about 3 seconds.

2 Personal regression example

In this example, the objective is to approximate a nonlinear function with a feedforward neural network. This function is unknown and must be approximated. A set of 13600 uniform samples is given (which are noise-free). An own individual dataset is created from this. It consists of X1, X2 and the target function *Tnew*. 3000 independent samples are drawn and split up into a training set, validation set, and test set. These sets have the following purpose [1]:

- Training set: the training set is a set of examples that is used to fit the parameters in the model. In this case, it is used to fit the connection weights in the neural network. The trained model produces a result, which is then compared with the target/label for each input vector in the training set. Based on this comparison, the parameters of the model are adjusted.
- Validation set: the validation set is a set of examples that provide an unbiased evaluation of the built model. The main purpose is for the regularization of the model by early stopping. When the error on the validation set no longer decreases, the training of the networks is stopped. It is also used to tune the model's hyperparameters (such as number of hidden units in the network).

- Test set: the test set is a dataset used to provide an unbiased evaluation of a final model fit on the training dataset.

The training set is visualized in Figure 3 using the `scatteredInterpolant` and `mesh` functions in Matlab.

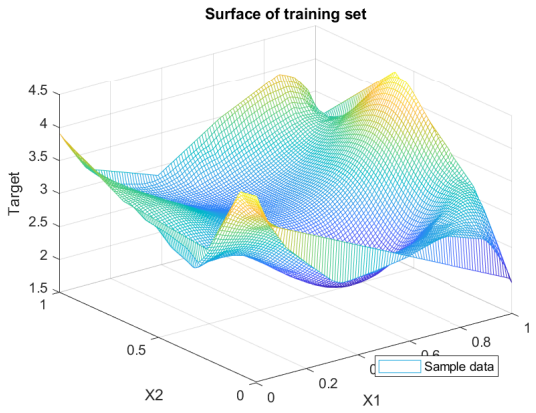
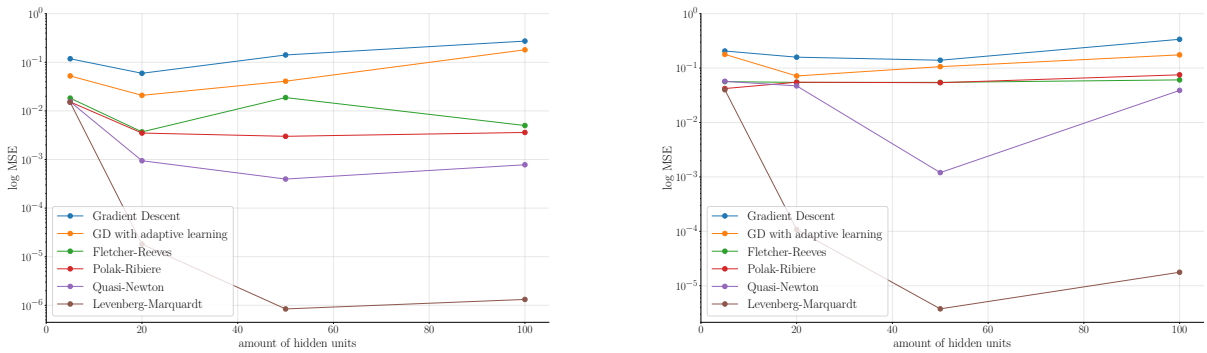


Figure 3: Surface of the training set.

In order to find the most suitable model for this problem, different models are trained where the number of hidden layers (1, 2), number of units per hidden layer (5, 20, 50, 100) and the training algorithm are changed. The model is evaluated based on the average MSE of the test set vs. the actual target value. The model is trained for 500 epochs. Figure 4 shows the results. The log of the MSE is plotted because of the different order of magnitude the MSE obtained with the training algorithms.

The best performance over the test set for 1 hidden layer, is obtained with 50 hidden units for most algorithms. For the gradient descent algorithms and the Fletcher-Reeves algorithm, the optimal number of hidden units is around 20. Increasing the number of hidden units to 100 does not produce better predictions. When models are trained with two hidden layers, the MSE increases for all training algorithms. Adding an extra layer does not seem to improve the performance. The optimal number of hidden neurons is around 50 for Levenberg-Marquardt and Quasi-Newton algorithms. For the other algorithms, 20 hidden units per layer seems the best. Overall, the Levenberg-Marquardt algorithm is the most suitable algorithm for this problem. The most optimal configuration seems to be 1 hidden layer with 50 units.



(a) one hidden layer

(b) two hidden layers

Figure 4: MSE vs. number of hidden neurons per layer. Note the logarithmic y-axis

Figure 5 presents the performance of the network with the optimal parameters (1 hidden layer with 50 units, LM-algorithm). An average MSE of $3.66E-6$ is achieved with this network. The performance could further be improved when more combinations of hidden layers and units are tested, to find the optimal combination.

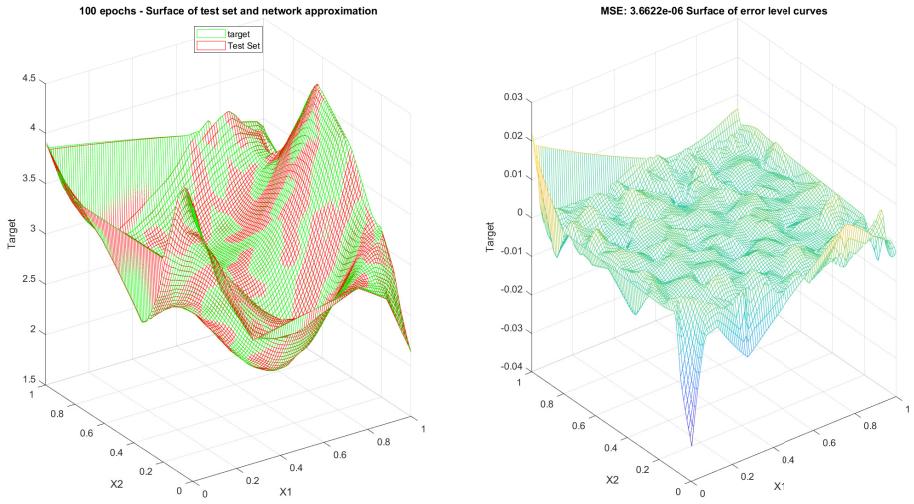
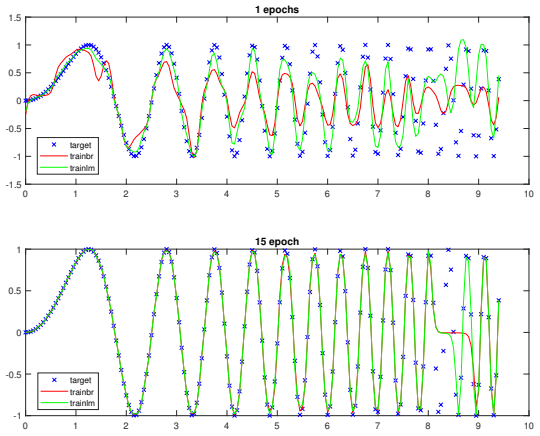


Figure 5: Left: Surface of the test set (red) vs. approximation by the neural network consisting of 1 hidden layer with 50 neurons (green). Right: Surface of the error level curves.

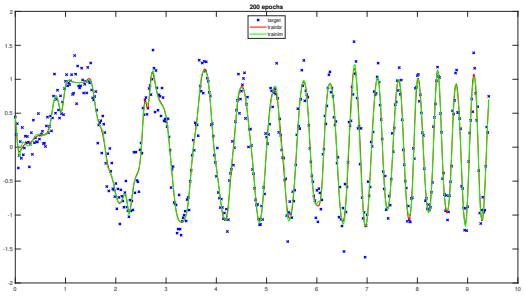
3 Bayesian inference of hyperparameters

In this section, Bayesian inference is used to find the hyperparameters of a model to approximate a sine function (same as in in Section 1). The Bayesian method has a good performance for approximating the sine function. When compared with the Levenberg-Marquardt method, it performs just slightly worse.

Now, noise is added to the sine function to reveal overfitting of training algorithms if it occurs. Figure 6b shows that both algorithms have a similar performance in approximating the noisy function. Both are robust against overfitting.



(a) noiseless function data



(b) noisy function data

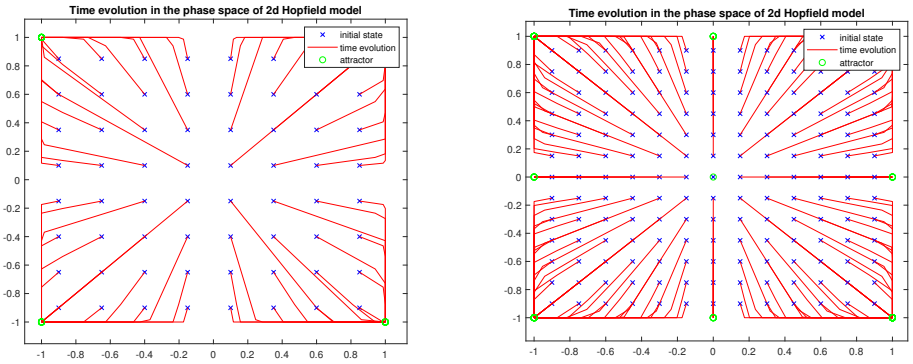
Figure 6: Function estimation with Bayesian method vs Levenberg-Marquardt method

References

[1] Training, validation, and test sets - Wikipedia.

1 Hopfield Network

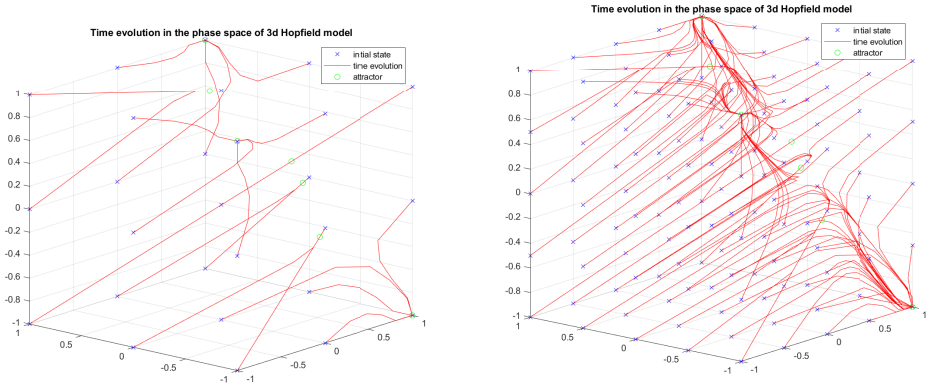
A Hopfield network is created with two neurons that has 3 attractors $T = [1\ 1; -1 - 1; 1 - 1]^T$. 49 points of high symmetry between (-1,-1) and (1,1) are initialized. Figure 1a shows the obtained attractors after 20 iterations. 3 of the 4 attractors are the ones used to create the network. There is one 'spurious' attractor at (-1, 1), which is an unwanted attractor. These spurious attractors are linear combinations of the stored patterns used for training. In this network, it takes up to 16 iterations for an initial point to reach the attractor. When we start from another initial vector with more points (169), one can see in Figure 1b that we get more spurious attractors in the field. In this network, it took 13 iterations maximally to reach an (unwanted) attractor. The attractors are 'stable' because once the values of the neurons correspond to these of an attractor, they do not change anymore when the network is updated.



(a) 1 unwanted attractor (b) 6 unwanted attractors

Figure 1: Obtained attractors of a 2D Hopfield network for various initial vectors

The same procedure is done now for the three neuron Hopfield network with three attractors $T = [1\ 1\ 1; -1 - 1\ 1; 1\ -1 - 1]^T$. Figure 2 shows the result. This time, there are three spurious attractors when initializing 27 symmetrical points in the space. When more points are initialized, no new unwanted attractors are found. With 3 neurons, it took up to 235 iterations for all initial points to reach an attractor. The more neuron the network has, the longer it takes for a vector to reach its stable state.



(a) 27 initial points, 3 unwanted attractors (b) Hopfield network with 6 unwanted attractors

Figure 2: Obtained attractors of a 3D Hopfield network for various initial vectors

1.1 Hopdigit function

The `hopdigit` function is a Hopfield network that is able to retrieve the patterns of handwritten digits out of noisy digits. The Hopfield model is not always able to reconstruct the correct digits when the level of noise is too high, regardless of the number of iterations. This is probably because the network cannot reach a stable state. When the noise level is acceptable, more iterations provide a higher chance of reconstruction of the correct digit. Also, if more iterations of the network are run, the model gives a clearer reconstruction of the digit.

2 Long Short-Term Memory Networks

2.1 Time series prediction with a neural net

In this section, we want to predict the next 100 points of the Santa Fe chaotic laser data. A multilayer perceptron with one hidden layer is trained on 1000 given data points, for 100 iterations. Afterwards, the predictive ability of the model is tested with a test set (which was not used during training). Different combination of lags and number of neurons are tried to tune the model. Table 1 lists the RMSE values for the different combinations. There is not really a clear relationship between RMSE and the lag value, or the RMSE and the number of neurons. However, a lag value of five and 10-20 neurons gives the best performance on the test set. Figure 3 shows the predictions of an MLP with these parameters. No optimal combination of parameters could be found that provided a better prediction of the final part of the laser function.

# neurons	lag 3	lag 5	lag 10	lag 20	lag 50
5	192	161	184	108	134
10	153	64	74	96	99
20	123	63	133	128	145
50	178	143	317	106	127
100	156	385	242	312	154

Table 1: RMSE values for different combinations of lag values and number of neurons in the hidden layer of an MLP

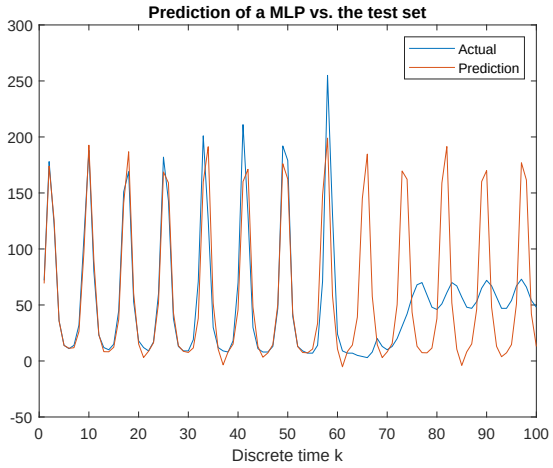


Figure 3: Prediction of a MLP vs the actual test data.

2.2 Long short-term memory network (LSTM)

Recurrent neural networks (RNN) suffer from short-term memory. If a sequence is long enough, they will have a hard time carrying information from earlier time steps to later ones. During back propagation, recurrent neural networks also suffer from the vanishing gradient problem. Gradients are values used to update the weights of a neural net. The vanishing gradient problem occurs when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it does not contribute too much learning. LSTMs were created as a solution to short-term memory. They are a special kind of recurrent neural networks which are capable of learning long term dependencies. The advantage of this type of network is that information can be stored in or read from an LSTM cell. These cells have internal gates that can regulate the flow of information. Figure 4 shows the structure. LSTM networks are very suitable to make predictions on time series data [1].

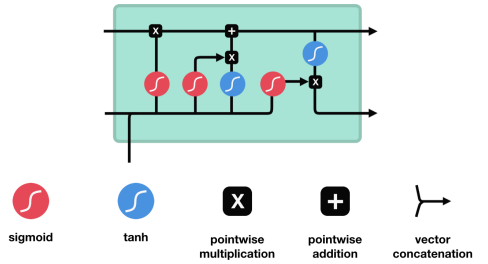


Figure 4: an LSTM cell and its operations [1]

There are some parameters that can be tuned in order to come up with an optimal model for predicting the Santa Fe dataset. These hyper-parameters include the number of neurons in the hidden layer, the initial learn rate, the factor with which the learn rate is dropped and the lag value. The following parameters were chosen:

hidden units: There is no fixed procedure to determine the optimal number of hidden units. More neurons in the hidden layer can increase the performance, but also increases the training time required to determine the weights. 5 values are tested: 10, 20, 50 and 100 units. **50 units** in the hidden layer resulted in the lowest RMSE for the predictions.

learning rate: Generally, a high initial learning rate allows the model to learn faster but yields a sub-optimal final set of weights. A small learning rate allows to model to learn an optimal set of weights, but it may take significantly longer to train. The optimal choice of the learning rate lies somewhere between these extremes. 4 values are tested: 0.001, 0.005, 0.01 and 0.05. An initial learning rate of **0.005** yields the lowest RMSE after 250 epochs of training.

drop factor: After certain periods of time, the learning rate is reduced in order to exploit the optimal weights at the end. The factor with which the (initial) learning rate is reduced is determined by the drop factor. When the drop factor is set low (0.1-0.2) and the initial learning rate is also low, the training is not efficient as there are only small improvements on the error during training. A high drop factor is best combined with a relatively low initial learning rate in order to have a large improvement during training. Because of this, the drop factor is set to **0.2** in combination with an initial learning rate of 0.005 and a drop period of 125.

lag: The lag value of the model is the amount of previous observed values that contributes to the next prediction. When the lag value is increased, more previous values will be used when making the next predictions. Lag values of 1, 10 and 25 were tested. A lag value of **10** seems to be optimal for the Santa Fe dataset.

Next, a model is trained with the training data (predictions in test data are not included) and the parameters above. With this model, the test set is predicted using the `predictAndUpdateState` function. This is shown in Figure 5. The graph at the top left shows the results when using the previous prediction as input for the next prediction (=lag of 1). The trained model at the top right with a lag value of 10 yields the lowest RMSE and makes predictions that are quite close to the actual values of the test set. The predicted waves after point 70 only come a bit too early but have to correct height. A lag value of 25 is too high, as can be seen in the graph at the bottom left.

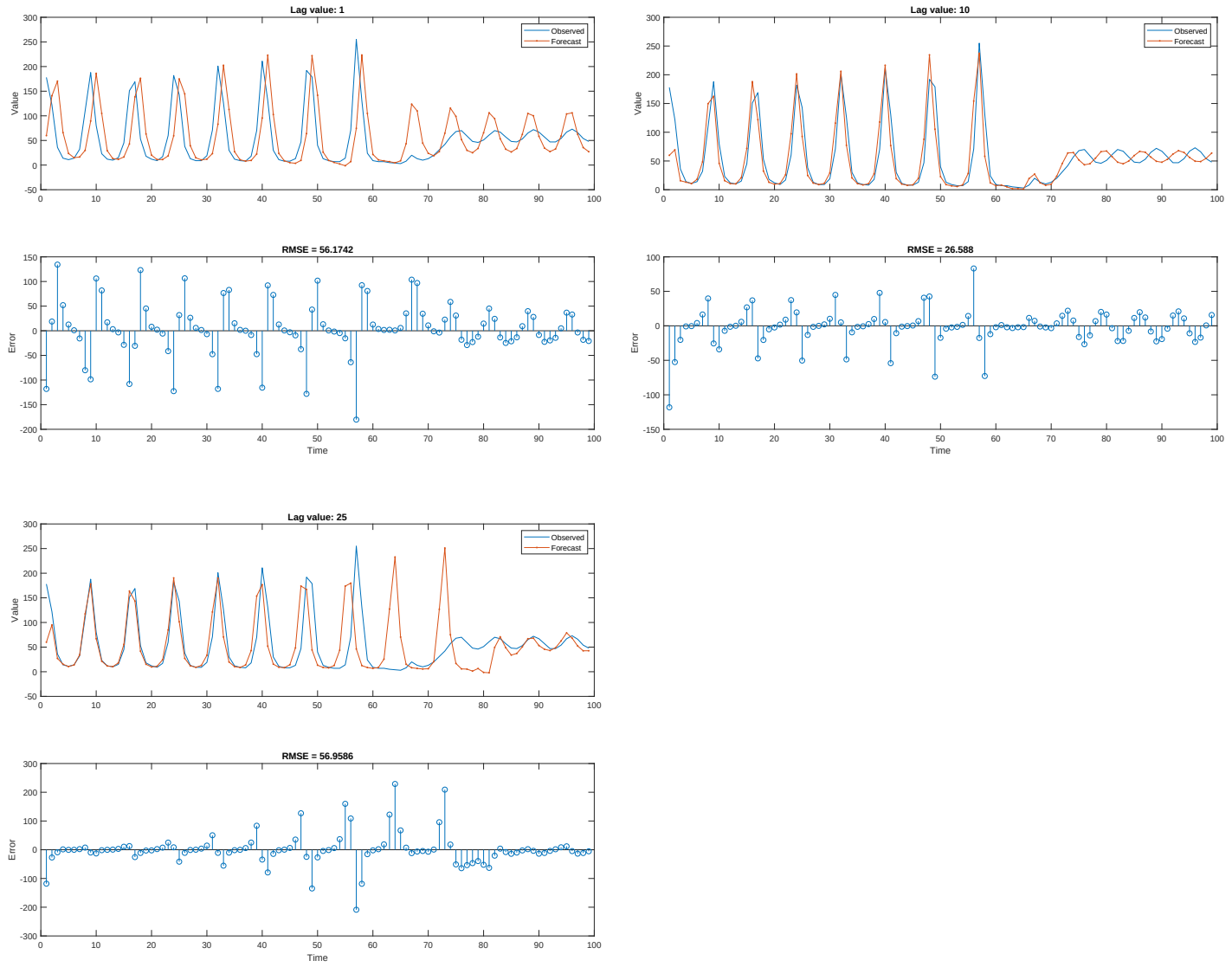


Figure 5: Predicted vs. observed values on the test dataset for different lag values (1-10-25 respectively)

The results from an LSTM network for a time series prediction is a lot better than the results of the 'recurrent' MLP. The recurrent MLP is in fact just a standard feedforward neural net, where the previous predictions are used as an input to predict the next observations. The LSTM network does a better job at predicting the test values. The average RMSE of the tuned MLP network is 63, while the average RMSE of the LSTM network is less than half, namely 27. The special structure of an LSTM network makes it possible to memorize past patterns and can memorize or forget new information. These networks can also deal with the vanishing gradient problem, in contrast to feedforward neural nets. Because of this, for time-series prediction, LSTM networks are preferred.

References

[1] PHI, M. Illustrated Guide to LSTM's and GRU's: A step by step explanation — Towards Data Science.

1 Principal Component Analysis

A dataset of handwritten images of the digit 3 is loaded and as a first exercise, the mean 3 is calculated with the `mean` function. After this, the covariance matrix of the whole dataset is computed, along with its eigenvalues and eigenvectors. As can be seen in Figure 1, most of the eigenvalues have a very small value close to 0. There are only about 50 out of 256 eigenvalues with a significant value. Because of this, the digits should not become unrecognizable when the dimensions are greatly reduced.

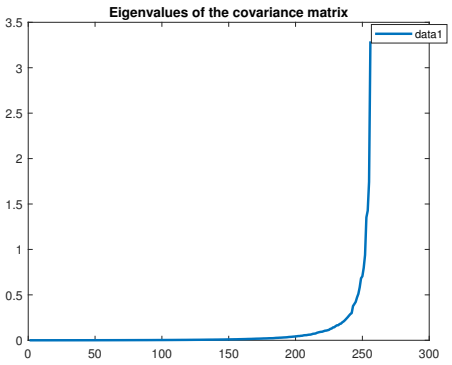


Figure 1: eigenvalues of the covariance matrix

Now, the dataset is compressed by projecting the images onto one, two, three and four principal components respectively. Afterwards, the original image is reconstructed. The results are shown in Figure 2. Reducing the dataset to one principal component is too much, but when reducing it to two principal components, a fairly recognizable 3 can already be reconstructed. When more principal components are kept, the quality of reconstruction gets better and better.

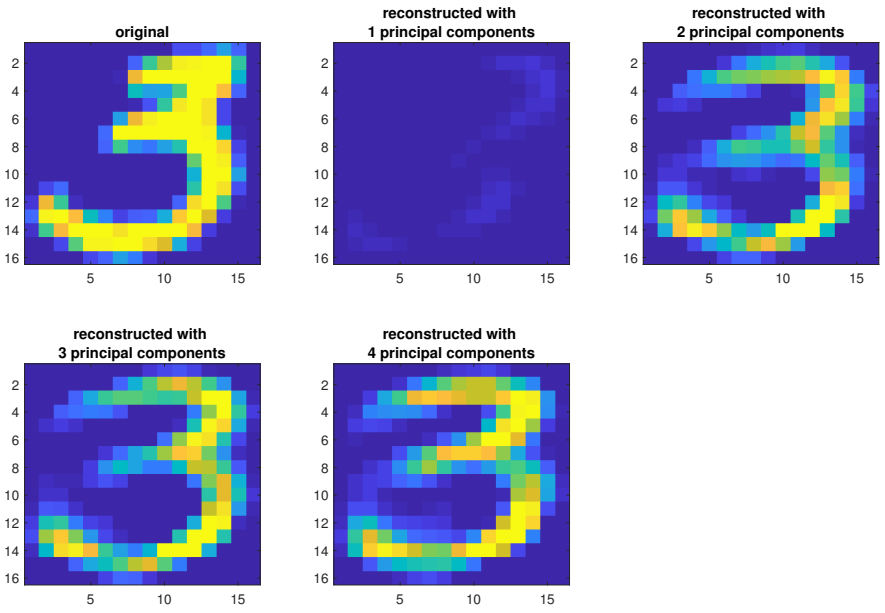
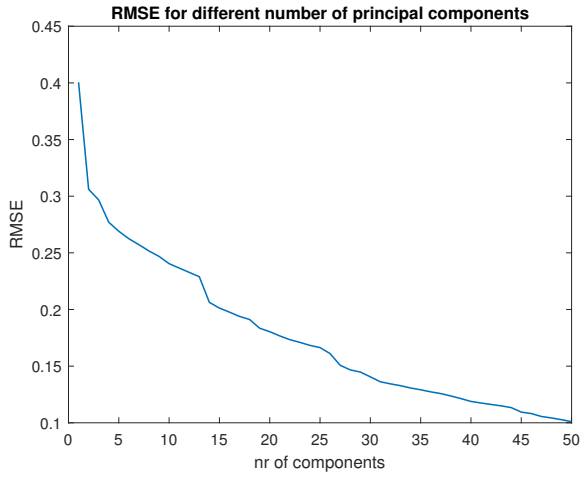
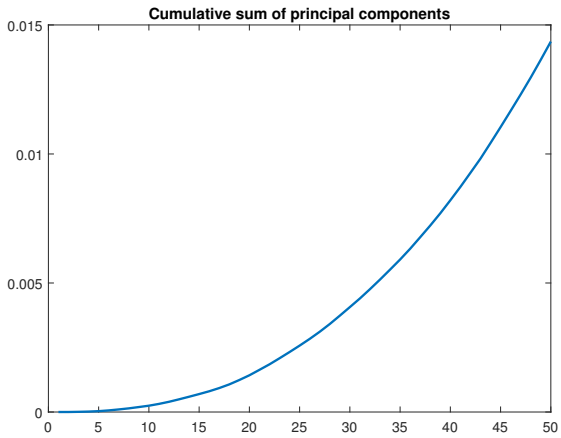


Figure 2: Reconstructed digit 3 after PCA

In the next step, a function is made that compresses the entire dataset by projecting it on q principal components. Then, the image is reconstructed and the RMSE of the reconstructed image vs. the original image is measured. This is done for values of q from 1 up to 50. Figure 3a shows how the reconstruction error decreases as the number of components increases. When q=256, the expected reconstruction error would be 0, as there are only 256 pixels in each image. However, there is a very small error of $8.82 \cdot 10^{-16}$. This is probably because of the floating-point operations, which works with a limited number of decimal places.
Figure 3b shows the cumulative sum of the 50 smallest eigenvalues. As can be seen, these values are very small. Leaving out the smallest eigenvalues has a minor effect on the error. The more eigenvalues are captured, the lower the error gets. Hence, the squared reconstruction error induced by not using a certain principal component is proportional to its eigenvalue.



(a) Reconstruction error vs. number of principal components



(b) Cumulative sum of 50 smallest eigenvalues

Figure 3

2 Stacked Autoencoders

An autoencoder is a type of network that can learn a representation (encoding) for a set of data. This representation can be used for dimensionality reduction, just like PCA. In this section, we perform fine tuning on the parameters of a stacked autoencoder network for digit classification. The network consists of two auto-encoders in the hidden layer and a softmax classifier layer. Afterwards, the performance is compared to a normal multilayer neural network. The performance of the different network architectures are compared in Table 1. Fine tuning has a very beneficial effect on the stacked autoencoders with two hidden layers. The error is reduced from 17.7% to 0.3%. A stacked autoencoder with two layers already performs better than the MLP model. Increasing the number of epochs for training did not lead further to a noticeably higher accuracy. This shows the strength of fine tuning.

Before fine tuning, the stacked auto-encoder consists of the weights of each auto-encoder that were trained individually. Also, the auto-encoders in the hidden layers were trained to estimate the input data, and are not optimized for the possible output values. Better results are obtained when fine tuning is applied after the first phase of training. It starts with the weights obtained by the individual autoencoders. Then, the knowledge of the labeled output values is provided and backpropagation is applied. The parameters of all layers are tuned at the same time.

	Stacked auto-encoder without fine tuning	Stacked auto-encoder with fine tuning	Multilayer neural network 1 hidden layer (100 units)	Multilayer neural network 2 hidden layers (100 and 50 units)
Error	17.7 %	0.3 %	3.8%	3.4%

Table 1: Error on the test set with different neural network architectures for digits classification

3 Convolutional Neural Networks (CNN)

In the last part of this assignment, a given CNN is used to do classification of the digits. In this paragraph, the questions in the assignment about the CNN are answered.

Questions:
"*Take a look at the first convolutional layer (layer 2) and at the dimension of the weights. What do these weights represent?*"

These weights represent the patterns for which the specific filter will be activated. For example, if the input image has a similar pattern, then the output of the filter will be high. This will lead to an activation. Conceptually, the convolutional layers act as feature detectors, while the weights define how these features look.

"*Inspect layers 1 to 5. If you know that a ReLU and a Cross Channel Normalization layer do not affect the dimension of the input, what is the dimension of the input at the start of layer 6 and why?*"

Before layer 6, There are two layers that affect the dimension of the input; a convolution layer and a pooling layer. The change in dimension that these layers cause, can be calculated as follows:

$$n_x \times n_y \times n_c \rightarrow \left(\frac{n_x - k_s}{s} + 1\right) \times \left(\frac{n_y - k_s}{s} + 1\right) \times n_f \qquad \text{with padding=0} \qquad (1)$$

With k_s the kernel size, s the stride and n_f the number of filters. The dimension of the input is (227x227x3). After the first convolution with 96 filters, kernel size 11 and stride 4, the dimension becomes (55x55x96). The same formula is now applied for the pooling layer, which keeps only the maximum value in the window. This layer has window size 3 and stride 2. This leads to the dimension (27x27x96) at the start of layer 6.

"*What is the dimension of the inputs before the final classification part of the network? How does this compare with the initial dimension? Briefly discuss the advantage of CNNs over fully connected networks for image classification.* "

The input of the fully connected layers has a size of 9216. This is significantly less that the input size of 227x227x3 (=154587). This is also the main advantage of CNNs. The number of parameters that needs to be trained are greatly reduced by exploiting the local connectivity of images.

Next, the script `CNNdigits.m` is run to investigate some CNN architectures. The filter size, number of filters and different number of layers are tested out. As a starting point, the default parameters are used (CNN 1). This architecture has an accuracy of 83,8 %, which is not optimal. In CNN 2, the filter is reduced to 3, which resulted in a model with a higher accuracy and faster training time. In CNN 3, more layers are added. In most cases, adding more convolution or softmax layers did not improve the accuracy. However, adding an extra convolution layer with a larger filter size before the fully connected layers improved the accuracy a little compared to the previous CNN. The best architecture (CNN 4) was obtained when doubling the number of filters in the convolution layers. This resulted in an accuracy of 97,3%, with the accuracy of the mini batches in the last iterations reaching 100%. The disadvantage is that the training time increases when more filters are used. The different architectures are compared in Table 2 on the following page.

CNN	1	2	3	4
layers	imageInputLayer([28 28 1]) convolution2dLayer(5,12) reluLayermaxPooling2dLayer(2,'Stride',2) convolution2dLayer(5,24) reluLayer fullyConnectedLayer(10) softmaxLayer classificationLayer()	imageInputLayer([28 28 1]) **convolution2dLayer(3,12)** reluLayermaxPooling2dLayer(2,'Stride',2) **convolution2dLayer(3,24)** reluLayer fullyConnectedLayer(10) softmaxLayer classificationLayer()	imageInputLayer([28 28 1]) convolution2dLayer(5,12) reluLayermaxPooling2dLayer(2,'Stride',2) convolution2dLayer(5,24) reluLayer **convolution2dLayer(5,48)** **reluLayer** fullyConnectedLayer(10) softmaxLayer classificationLayer()	imageInputLayer([28 28 1]) **convolution2dLayer(5,24)** reluLayermaxPooling2dLayer(2,'Stride',2) **convolution2dLayer(5,48)** reluLayer fullyConnectedLayer(10) softmaxLayer classificationLayer()
training time	34.85	28.8	43.8	36.56229
accuracy	0.8380	0.9104	0.9236	0.9728

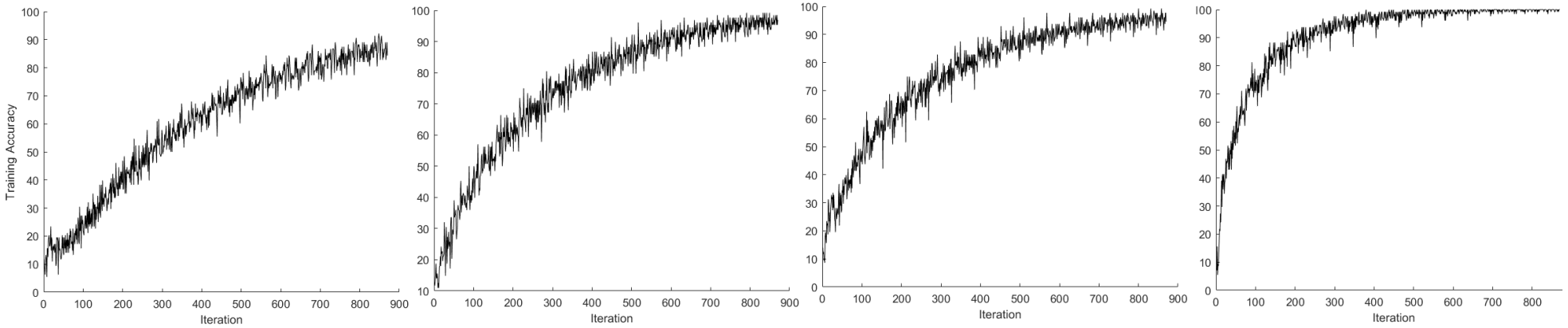
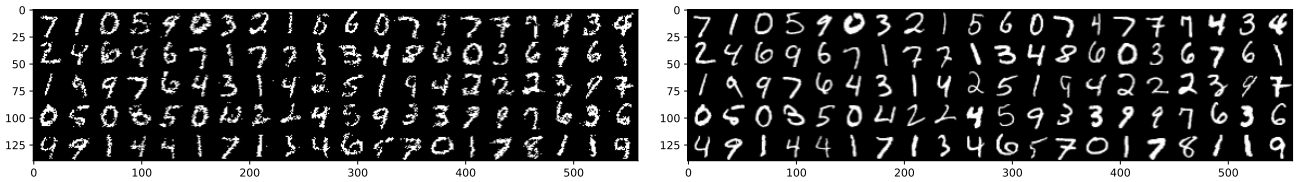


Table 2: Comparison different CNN architectures. Difference in architecture compared to CNN 1 are marked in bold.

1 Restricted Boltzmann Machines (RBM)

There are some different parameters that comes into play when training a RBM neural net. In order to optimize the model, the number of epochs and number of components are tuned.

The *number of components* parameter defines the number of binary hidden units in the network. The more hidden units, the better the model seems to perform, at the cost of a longer training time. The better performance is probably because the model can learn more patterns from the training set with more units. The more iterations, the longer the model takes to train, but the better the performance.
The *number of epochs* parameter controls the number of iterations/sweeps over the training set to perform during training.
Another hyper-parameter that has to be tuned is the *learning rate*. It controls how quickly the model is adapted to the problem. A high learning rate results in faster learning but may yield a set of weights that is sub-optimal. A smaller learning rate requires more training epochs but can give a better set of weights afterwards [1]. Some different combinations of these parameters are manually set in order to come up with a good model that can be trained in a reasonable time. The best model that is used further has the following parameters: 100 hidden units, learning rate 0.02, 10 epochs and took 3 minutes to train. After training, the performance of the network is evaluated visually by reconstructing unseen test images. The number of Gibbs sampling steps is set to 1. As can be seen in Figure 1 the RBM network is able to generate realistic digits based on the unseen test input digits.



(a) Sampled digits

(b) Real test digits

Figure 1: Generated images from the RBM net vs. unseen input images from the test set

Next, the RBM is used to reconstruct missing parts of unseen images. The more rows are removed, the more difficult it is for the RBM to create a recognizable reconstruction. An increased number of Gibbs steps improves the reconstruction, but when it is set too high (> 20), the images get corrupted. When the number of Gibbs steps is set to 1000, almost all generated images are a '1' or '7'.
For this case, the Gibbs steps is set to 12. The role of the number of hidden units, learning rate and iterations is described above. When these parameters are well chosen, the network will perform better in reconstructing the images. The location of the removed roles also plays a role. When characteristic parts of the digit are removed, the reconstruction is worse (e.g., the top horizontal part of '7' or the middle part of '0').

(a) five and ten **middle** rows removed respectively

(b) ten and fifteen **top** rows removed respectively

Figure 2: Unseen images with removed rows on top and reconstructed images at the bottom. Quality of reconstruction depends on the number of removed rows and the location of the removed rows in the digit.

2 Deep Boltzmann Machines (DBM)

Deep Boltzmann Machines are similar to RBMs. They also have visible layer but have multiple hidden layers. DBM can be interpreted as a series of RBMs stacked on each other. The left image in Figure 3 shows the filters extracted from the RBM. These look similar to the filters of the first layer of the DBM (middle image). They extract more simple features like edges. The filters of the second layer of the DBM already detect more refined features, like a specific shape of an image.

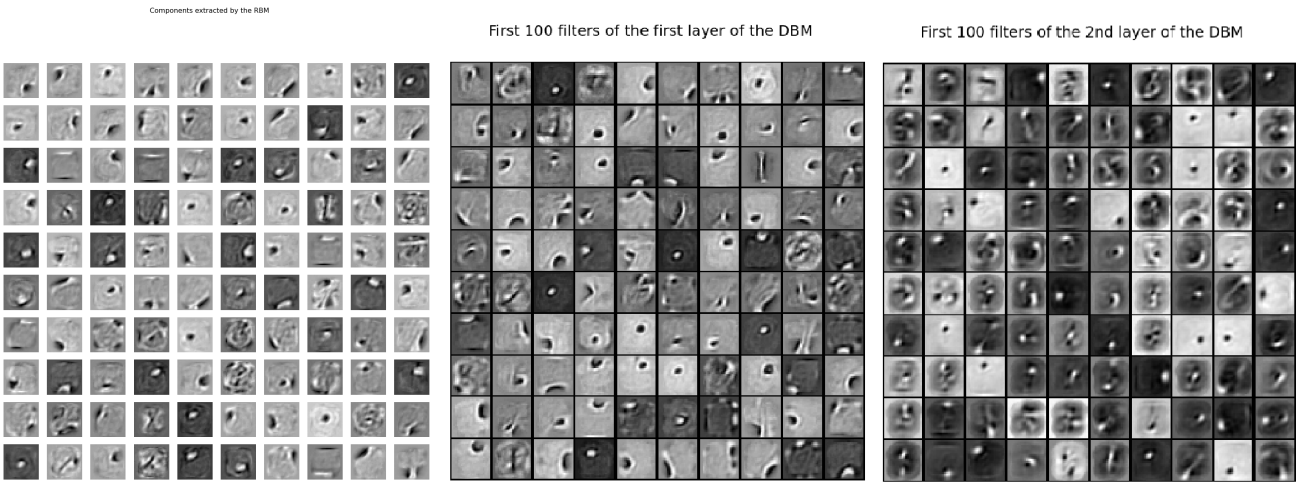


Figure 3: Extracted weights (filters) from the RBM and the DBM

The samples that are generated by the DBM look clearly better that these from the RBM. The digits look sharp and realistic, while the digits generated by the RBM look more sketchy. This is because the structure of the DBM has more layers. These hidden layers act as feature detectors on top of each other and the network can extract more high-level features. On the other hand, the RBM architecture is a shallow architecture with less components, which make it more difficult to produce high-quality digits. However, not all generated images from the DBM are recognizable as a digit, as can be seen in Figure 4 (row 3, column 5).

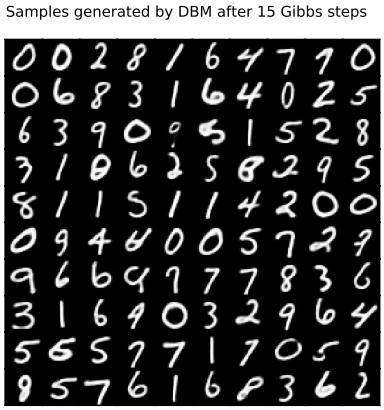


Figure 4

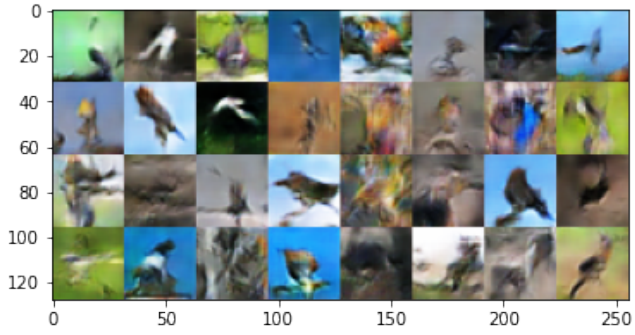
3 Generative Adversarial Networks

A Generative Adversarial Network (GAN) can be difficult to train, because the both the generator and the discriminator has to learn at the same time. Because of this, the training is not stable. Radford et al. [3] propose a solution for this instability problem with a Deep Convolutional GAN (DCGAN). They make the following recommendations on the architecture of their proposed network:

- Replace any pooling layers with (fractional) strided convolutions
- Use batch normalization in both the generator and the discriminator
- Remove fully connected hidden layers for deeper architectures
- Use ReLU activation in generator for all layers except for the output, which uses Tanh
- Use LeakyReLU activation in the discriminator for all layers

The `train_random_batches` function from the given code takes these guidelines into account. A DCGAN network is trained on a CIFAR-10 dataset containing images of birds. The network is trained for 20000 batches with a batch size of 32. Figure 5b shows the progress of loss and accuracy of the generator and discriminator, with a moving average of 50. The training is relatively stable. Initially, the accuracy of the generator is low, and its loss is high. The accuracy of the discriminator is high, and its loss is low. This is because the generated images by the generator are bad in the beginning, and the discriminator can easily see that the image is not real. During training, the generator suddenly improves a lot, which causes a drop in the accuracy of the discriminator. Afterwards, the accuracy of both the discriminator and generator gradually improve a little. After around 10000 iterations of training, the loss and accuracy of the discriminator in balance. After this point, the accuracy of the discriminator increases, and the accuracy of the generator decreases, so that the balance gets lost again.

Figure 5a shows the images that the network generated after training. These images resemble birds quite well; however, they are not very detailed. Probably because the images of birds the net was trained on were already quite blurry.



(a) Generated images from the DCGAN

(b) Loss and accuracy of the generator vs. discriminator during training

Figure 5

4 Optimal Transport (OT)

Figure 6 shows two images of a city characterized by their red and blue houses respectively. We want to change the colours of the second picture to the colours of the first picture and vice versa, while keeping the same structure. In the given code, OT is used to transfer the palette of colours between the two images. This is better than just swapping the pixels because the color intensities would not match. Each

pixel has an RGB value, and one can make histograms of the intensities of each channel. With OT, the shape of the histogram of the red channel is put into the shape of the histogram of the blue channel. In order to do this in an optimal way, the theory of Optimal Transport defines a total cost function that has to be minimized [2]:

$$C(T) = \sum_{k=1}^{n} c(x_k, T(x_k)) \qquad \text{with c a cost function and T the mapping from } x_k \text{ to y}$$

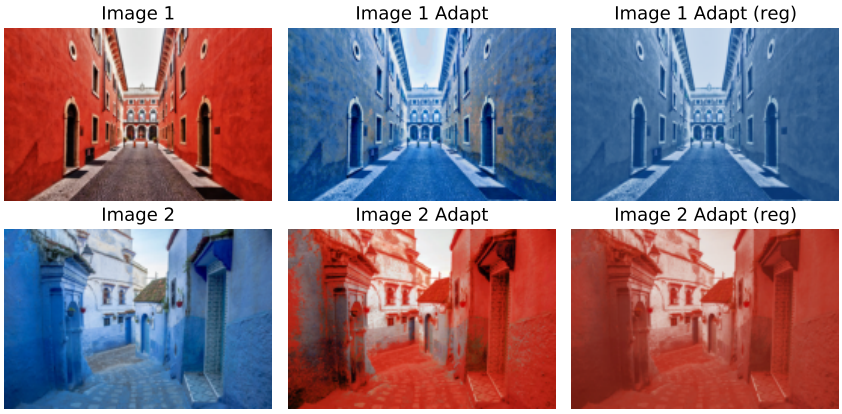
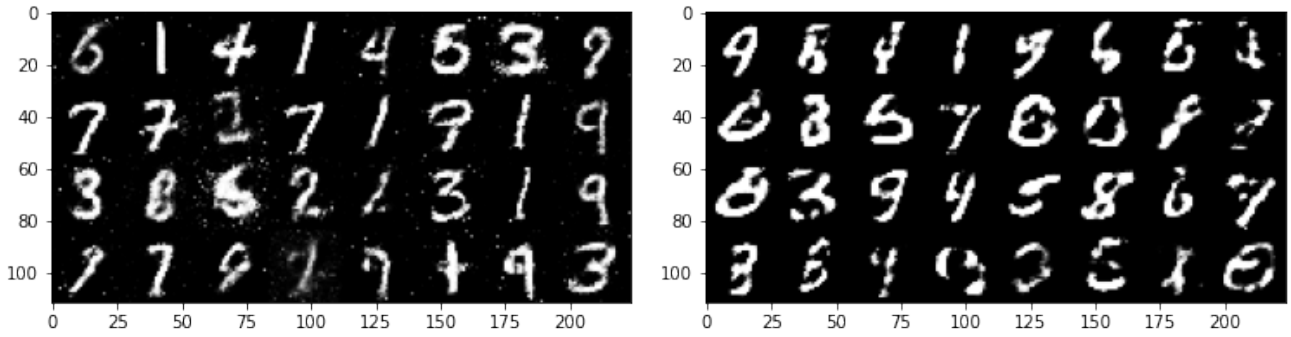


Figure 6: Optimal Transport used to swap colours between image 1 and image 2

The optimal transport did indeed swap the colours, while the structure and the intensities in the images remain unaffected.

Optimal transport is also used with GANs. In this paragraph, a standard GAN and a Wasserstein GAN is trained (20000 batches). Generated images from both networks are compared in Figure 7. The Wasserstein GAN is more stable during training as it produces clearer and less noisy images. However, the accuracy of the discriminator and generator is lower, and some of the generated digits are corrupted.



(a) Generated images from the standard GAN

(b) Generated images from the Wasserstein GAN

Figure 7: Generated images after 20000 batches of training

References

[1] BROWNLEE, J. Understand the Impact of Learning Rate on Neural Network Performance, 2019.

[2] BROWNLEE, J. Understand the Impact of Learning Rate on Neural Network Performance, 2019.

[3] RADFORD, A., METZ, L., AND CHINTALA, S. Unsupervised representation learning with deep convolutional generative adversarial networks.