KATHOLIEKE UNIVERSITEIT LEUVEN

Prof. Dr. ir. Johan A. K. Suykens

Artificial Neural Networks & Deep Learning

Exercise Reports

Marco Bischoff (R1012984)

May 29, 2024

Contents

1 Supervised Learning and Generalization **3**
1.3 In the Wide Jungle of the Training 3
1.4 A Personal Regression Exercise 4

2 Recurrent Neural Networks **6**
2.1 Hopfield Networks 6
2.2 Timeseries Prediction 8

3 Deep Feature Learning **9**
3.1 Stacked Autoencoders 9
3.2 Convolutional Neural Networks 10
3.3 Self-Attention and Transformers 11

4 Generative Models **13**
4.1 Energy-Based Models 13
4.2 Generator and Discriminator in the Ring 16
4.3 An Auto-Encoder With a Touch 17

1 Supervised Learning and Generalization

1.3 In the Wide Jungle of the Training

Task 1.3.1

The noise parameter controls the deviation of the training data from the true function. Figure 1 shows the impact of noise on the optimization process. For $noise = 0$, the data exactly matches the true function and the model will converge to the true function quickly. For $noise > 0$, the data is perturbed by noise and the model will take longer to converge. For $noise = 1$, the data is completely random and the model will only converge to the mean of the training data without being able to capture the underlying function.

Task 1.3.2

Vanilla gradient descent is the slowest of the three methods. It computes the gradient of the loss function for the entire training set at each iteration. Stochastic gradient descent (SGD) is faster than vanilla gradient descent, because it computes the gradient for a random subset of the training data at each iteration. However it has a higher variance in the loss function. Accelerated gradient descent is the fastest of the three methods. It uses a momentum term to speed up convergence and reduce oscillations in the loss function.

Task 1.3.3

For small networks, vanilla gradient descent is sufficient, because the computation of the gradient is not very expensive. For larger networks, SGD is more appropriate due to its lower computational cost. Accelerated gradient descent is the best choice for very large networks, because it converges faster than the other two methods.

Task 1.3.4

The model is trained for 2500 epochs. In Figure 2, we can see that SGD with a learning rate of 0.05 and without momentum has an average training time, but very slow convergence. Using a learning rate of 0.1 already converges much faster, but also takes longer to compute. Momentum has a similar convergence rate and is much quicker to compute, but also has high variance. The Adam and the LBFGS optimizers converge the fastest, but LBFGS has the longest computational time. The Adam optimizer is the best choice for this problem, because it converges quickly and has low variance. All optimizers except for vanilla SGD with learning rate 0.05 can be considered to have converged after at most 1000 epochs.

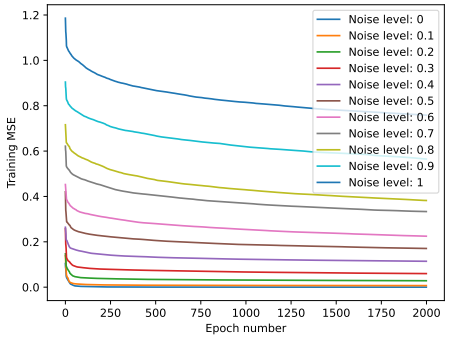


Figure 1: Impact of noise on the optimization process

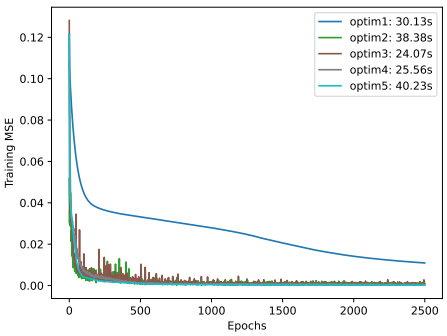


Figure 2: Comparison of optimizers

A bigger model

Task 1.3.5

The model has 34826 parameters in total as shown in Table 1.

Layer Type	Shape	# Param
(Input)	(28, 28, 1)	0
Conv2D	(26, 26, 32)	320
MaxPooling2D	(13, 13, 32)	0
Conv2D	(11, 11, 64)	18496
MaxPooling2D	(5, 5, 64)	0
Flatten	(1600,)	0
Dropout	(1600,)	0
Dense	(10,)	16010
Total		**34826**

Table 1: Model parameters

Task 1.3.6

The SGD optimizer has a much slower convergence rate than the Adam optimizer as shown in Figure 3. The Adadelta optimizer achieves very good performance after the first epoch already and has a low variance. The Adam optimizer is in between the two in terms of convergence rate and variance. Compared to the Adam optimizer, the Adadelta optimizer does not require a learning rate to be set, because it uses the gradient and the average of the squared gradient over a window of time steps to adapt the learning rate.

1.4 A Personal Regression Exercise

Task 1.4.1

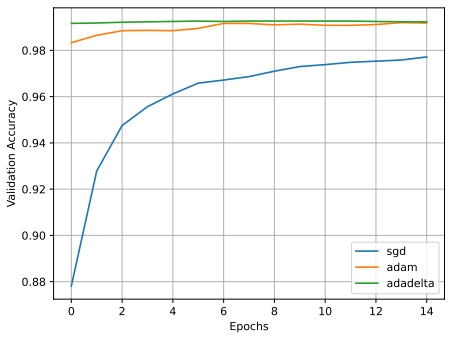


Figure 3: Validation accuracy

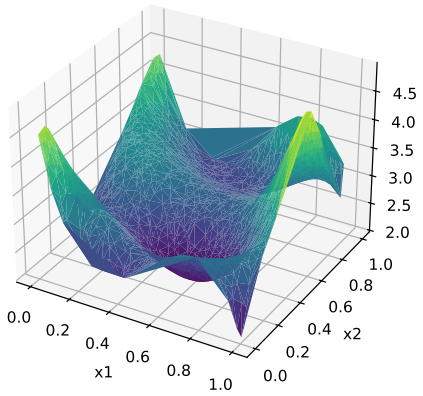


Figure 4: Function surface of the training data

If the same dataset is used for training and testing, the model will overfit the data and not generalize well to unseen data. Moreover, the evaluation of the model will be biased, because the model has already seen the test data during training. If two independent datasets are used, the performance on the test data corresponds more to real-world performance, where the model

has not seen the data before. Also, the performance on the test data will be bad, if the model has overfit the training data. The surface associated to the training set is shown in Figure 4.

Task 1.4.2

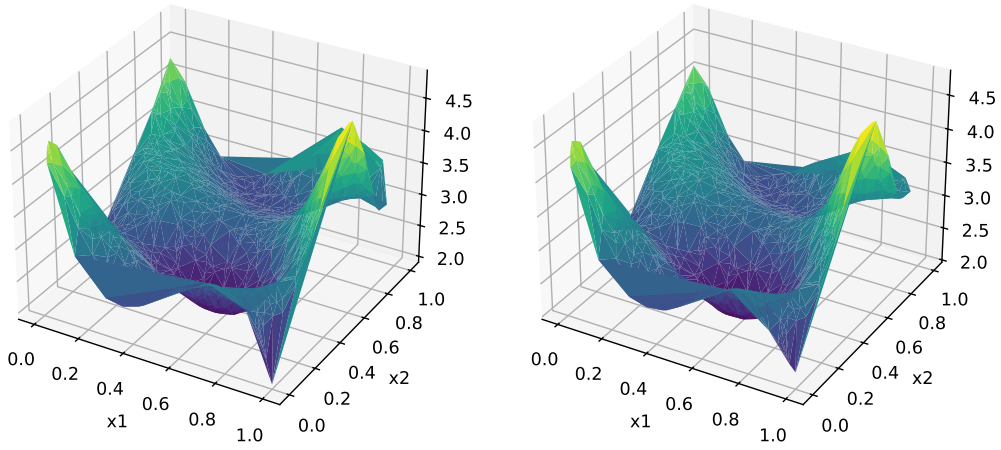
The model architecture is shown in Table 2. The choice of the activation functions seemed to have the biggest impact on the performance. The `mish` activation function performed much better than the other functions I tried. Increasing the number of layers and neurons also improved the performance. However, the model was overfitting the training data when using too many neurons. The choice of the optimizer and the learning rate did not have a big impact on the performance. Here, I used the Adam optimizer with a learning rate of 0.05. All choices were validated by calculating the mean squared error on the test data.

Layer Type	Shape	Activation Function	# Param
Dense	(16)	mish	48
Dense	(16)	mish	272
Dense	(16)	tanh	272
Dense	(1)	(None)	17
Total			**609**

Table 2: Regression model parameters

Task 1.4.3

The surfaces of the test data and the predicted data are shown in Figure 5. The model has already converged and further training would not improve the performance. The final mean squared error on the test data is 0.0017132.



(a) Test data (b) Predicted

Figure 5: Function surface

Task 1.4.4

I used early stopping to avoid overfitting. The training was stopped when the validation loss did not improve for 10 epochs. Another strategy to avoid overfitting is to use dropout layers, which sets a few neurons to zero during training.

2 Recurrent Neural Networks

2.1 Hopfield Networks

Task 2.1.1

All of the attractor states $[1, 1]$, $[-1, -1]$ and $[1, -1]$ are reached after a few iterations, as shown in Figure 6. We also get the unwanted attractor $[-1, 1]$ because the network is not able to distinguish between the patterns $[1, -1]$ and $[-1, 1]$. The ten random points in Figure 6 converged after an average of 8.6 iterations. The attractors are stable, as further iterations do not change the state of the network. However, if initial points are placed exactly in the middle between two attractors, they stay unaffected by updates, as shown in Figure 7. These points are additional attractors, which are not part of the target patterns. They are unstable, as small perturbations will make the network converge to one of the stable attractors.

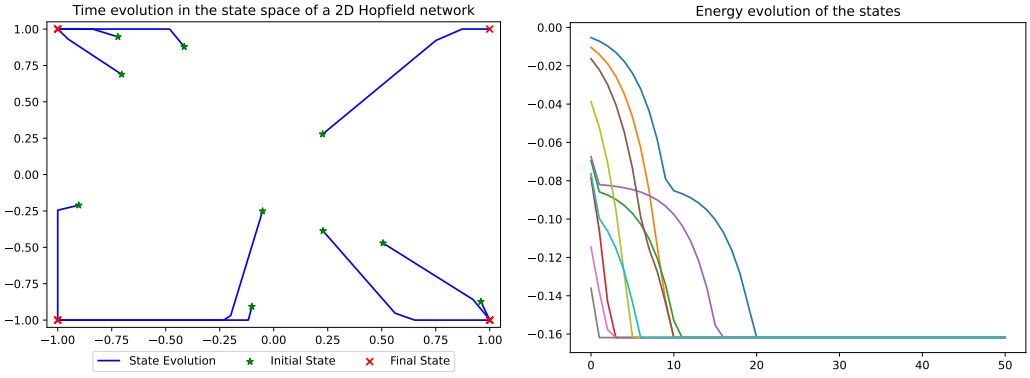


Figure 6: 2D network: random inputs

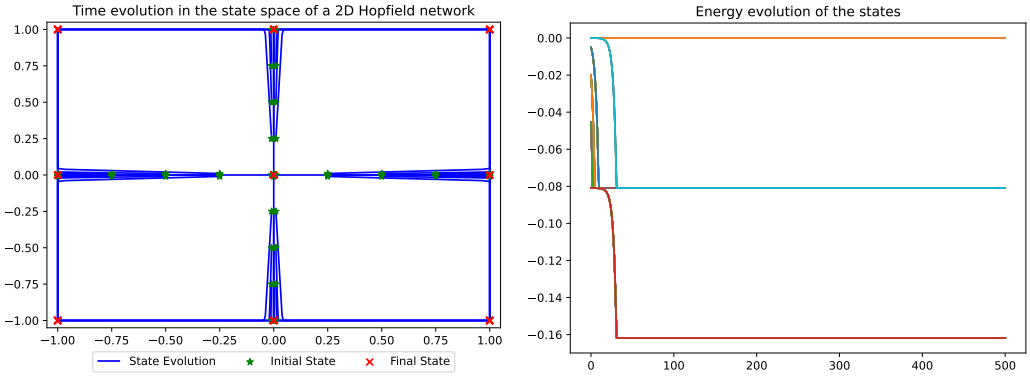


Figure 7: 2D network: inputs close to local minima

Task 2.1.2

For the 3D network, all the target patterns $[1, 1, 1]$, $[-1, -1, 1]$ and $[1, -1, -1]$ are reached, as shown in Figure 8. No additional attractors are found. The average number of iterations is 11.7. All of the attractors are stable, even if the initial points are placed exactly in the middle between

two attractors, as shown in Figure 9. The simulated points first move towards the plane that goes through all three attractors, and then moves along this plane to the closest attractor.

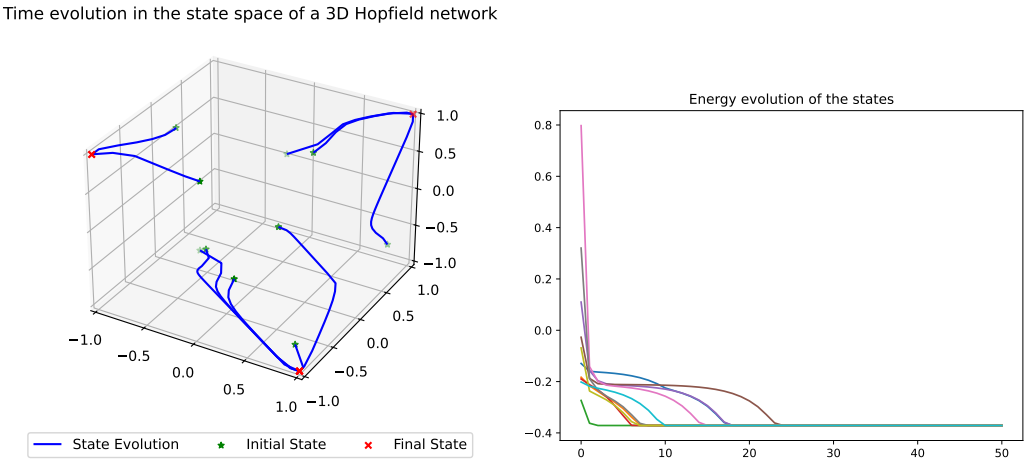


Figure 8: 3D network: random inputs

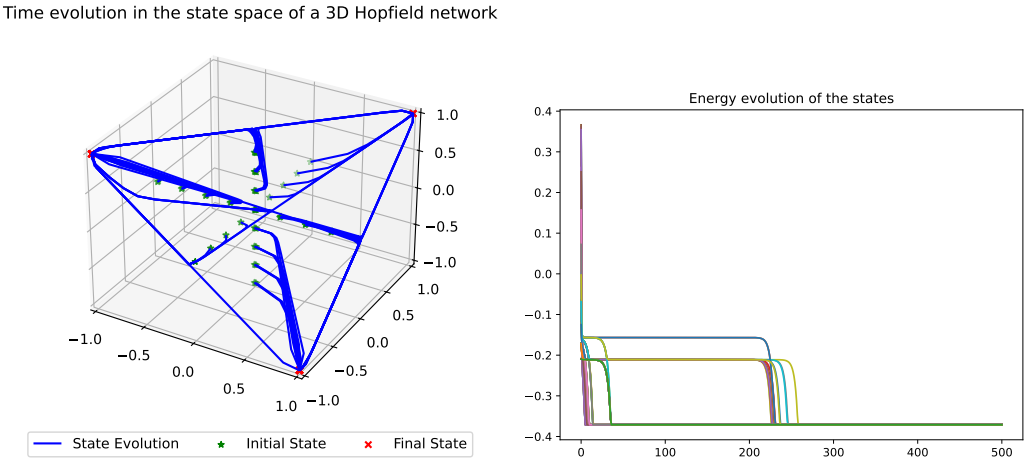


Figure 9: 3D network: inputs close to local minima

Task 2.1.3

For 100 iterations, the network is able to reconstruct the noisy digits for noise levels up to 5. Above this level, a few digits converge to the wrong attractor. Some digits are less robust to noise than others. For example, the digit 7 is often reconstructed as a 3. Also, with less iterations, the network often does not fully converge to an attractor. This leaves traces of the noisy input in the reconstructed digit. For a good reconstruction with a noise level of 5, the number of iterations should be at least 40. For a noise level of 1, the network is able to converge after only 4 iterations.

2.2 Timeseries Prediction

Task 2.2.1

The MLP model consists of an input layer with $lag = 80$ neurons, a hidden layer with 100 neurons, and an output layer with 1 neuron. The model is trained for 3 folds and 200 epochs each. The loss for each fold is shown in Figure 10. The parameters were tuned based on the loss on the validation set of size 150. Using a bigger hidden layer and more lag improved the performance the most. However, the model was overfitting a lot if there were too many validation folds. Using fewer folds with a bigger hidden layer yielded better results than using more folds with a smaller hidden layer. Also, the MSE on the test set would vary a lot between different runs with the same parameters. This could be due to the initialization of the weights. The final MSE on the test set was 277.3 for $lag = 80$ and 100 neurons in the hidden layer.

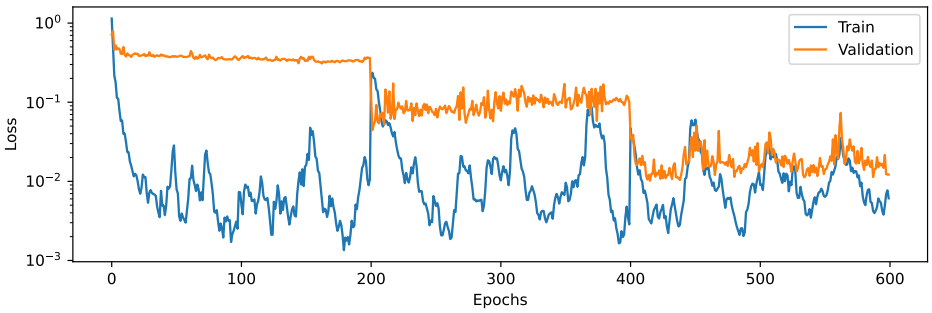


Figure 10: MLP loss for 3 folds and 200 epochs each.

Task 2.2.2

The LSTM model was also trained for 3 folds, $lag = 80$ and 100 neurons in the hidden layer, but only for 150 epochs. This was done to keep the results comparable to the MLP model. The parameters were tuned based on the loss on the validation set of size 150. The loss for each fold is shown in Figure 11, which still shows a small performance improvement for the third fold. When using more folds, the performance of the LSTM model was decreasing. Using a lower lag value also decreased the performance, which means that the LSTM model is able to capture a lot of the temporal dependencies. The final MSE on the test set was 90.3.

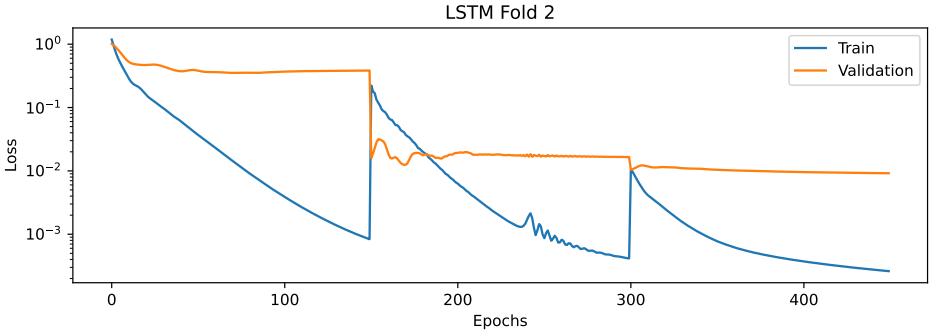


Figure 11: LSTM loss for 3 folds and 150 epochs each.

Task 2.2.3

As shown in Figure 12, the LSTM model outperforms the MLP model. For the first two thrids of the test set, the MLP and LSTM models predict the timeseries similarly. However, after the regular pattern changes, the MLP model is not able to predict the timeseries well. This could be due to the LSTM model being better at remembering and understanding the previous states of the timeseries. This is also reflected in the lower MSE of the LSTM model. Therefore, the LSTM model is preferred over the MLP model for this task.

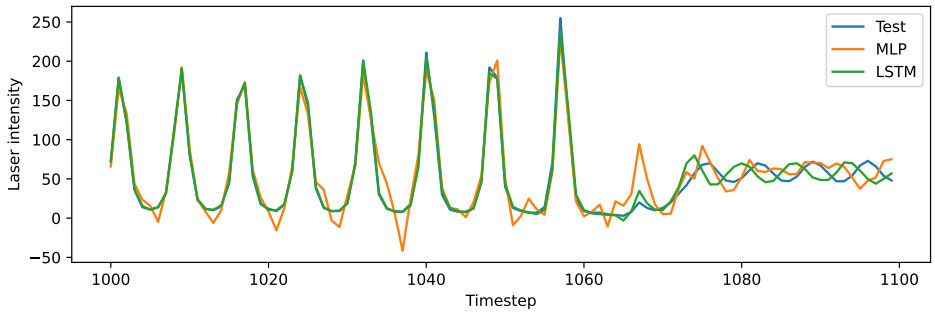


Figure 12: Prediction results on continuation of the Santa Fe laser dataset.

3 Deep Feature Learning

3.1 Stacked Autoencoders

Task 3.1.1

I am using two autoencoders stacked on top of each other with hidden dimensions 256 and 64 respectively. Changing the size of the network architecture did not yield significant improvements. Changing the number of epochs however did. I got the best results with 20 epochs for pre-training each layer, 30 epochs to train the classification layer and 50 epochs to fine-tune the whole network. The loss for every training step is shown in Figure 13. Fine-tuning significantly improved the performance. The accuracy on the test set was 0.955. Pretraining the network layer by layer helps to find a good initialization for the weights. This seems to work well in this case. However, the network was not able to learn by training the classification layer.

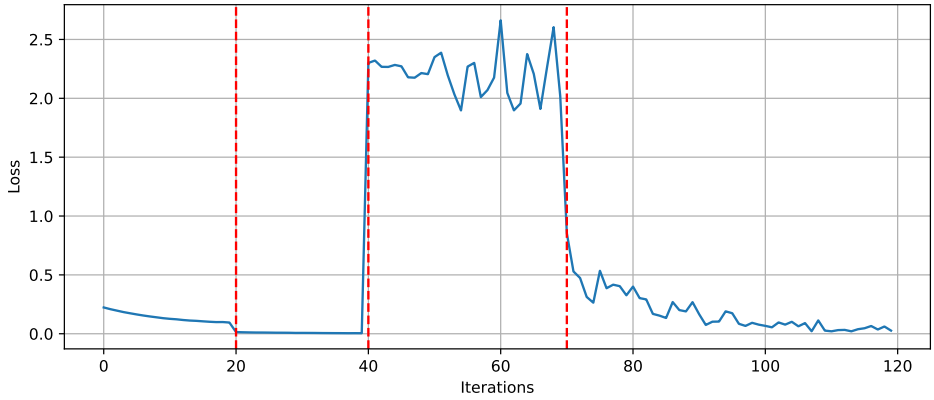


Figure 13: Loss for the stacked autoencoder. (1) Pre-training the first layer, (2) pre-training the second layer, (3) training the classification layer, (4) fine-tuning the whole network.

3.2 Convolutional Neural Networks

Task 3.2.1

(a) The output of the convolution is calculated as follows:

$$\mathbf{X} * \mathbf{K} = \begin{bmatrix} 2 & 5 & 4 & 1 \\ 3 & 1 & 2 & 0 \\ 4 & 5 & 7 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} 2*1+5*0+3*0+1*1 & 4*1+1*0+2*0+0*1 \\ 4*1+5*0+1*0+2*1 & 7*1+1*0+3*0+4*1 \end{bmatrix} \tag{2}$$

$$= \begin{bmatrix} 3 & 4 \\ 6 & 11 \end{bmatrix} \tag{3}$$

(b) Let $X \in \mathbb{R}^{n \times n}$ be the input matrix, $K \in \mathbb{R}^{m \times m}$ the kernel, $P \in \mathbb{N}$ the padding, $S \in \mathbb{N}$ the stride and $Y \in \mathbb{R}^{k \times k}$ the output matrix. Then the dimensionality of the output is determined by

$$k = \left\lfloor \frac{n - m + 2P}{S} \right\rfloor + 1 \tag{4}$$

(c) The big benefit is that, compared to fully connected networks, CNNs have way fewer parameters. This is because the weights are shared across the input. It also allows the network to capture how pixels are spatially related to each other. This is especially useful for image data.

Task 3.2.2

The final architecture has the following layers:

1. **Convolutional layer**
   - Number of filters: 16
   - Kernel size: $(4, 4)$
   - Stride: $(2, 2)$
   - Padding: $(2, 2)$
   - Batch normalization
   - ReLU activation
   - Dropout with probability 0.2
2. **Convolutional layer**
   - Number of filters: 32
   - Kernel size: $(4, 4)$
   - Stride: $(2, 2)$
   - Padding: $(1, 1)$
   - Batch normalization
   - ReLU activation
   - Dropout with probability 0.4
3. **Convolutional layer**
   - Number of filters: 32
   - Kernel size: $(3, 3)$
   - Stride: $(1, 1)$
   - Padding: $(1, 1)$
   - Batch normalization
   - ReLU activation
   - Dropout with probability 0.4
4. **Convolutional layer**
   - Number of filters: 64
   - Kernel size: $(2, 2)$
   - Stride: $(1, 1)$
   - Batch normalization
   - ReLU activation
5. **Fully connected layer** with 2304 inputs and 100 outputs
6. **Fully connected layer** with 100 inputs and 10 outputs

This architecture is the result of various experiments. I tested different output channels, kernel sizes, strides, padding, activation functions, dropout rates, batch normalization and pooling. The most important factors for the performance were the number of filters, bigger kernel sizes and dropout.
Adding more layers did not improve the performance. Too many parameters would lead to overfitting. This was also the case with bigger kernel sizes, but dropout helped to prevent this. Without dropout, the accuracy increased fast but stagnated quickly. Pooling increased the variance in the accuracy a lot. I tested the activation functions ReLU, Sigmoid and Tanh. ReLU performed the best. Batch normalization only improved the performance a little. The results of different kernel sizes, strides and padding were hard to interpret. The results were inconsistent, but had a small tendency that suggested that using a bigger kernel size in the first layers and smaller kernel sizes in the following layers worked best. To prevent too many parameters, I used strides of 2 in the first layers. Test accuracy was used as the main metric to evaluate the performance. The architecture above achieved an accuracy of 99.0%.

3.3 Self-Attention and Transformers

Task 3.3.1: Self-Attention Mechanism

The self-attention mechanism was introduced by Vaswani et al. in 2017, where they define the attention function for a set of queries.
Let $\mathbf{X} \in \mathbb{R}^{N \times d_e}$ be the input, where N is the number of tokens and d_e is the dimension of the word embeddings. Let $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{d_e \times d_k}$ and $\mathbf{W}_v \in \mathbb{R}^{d_e \times d_v}$ be the weight matrices for the queries, keys and values respectively. The dimensions of the queries, keys and values are related as follows:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \in \mathbb{R}^{N \times d_k}, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_k \in \mathbb{R}^{N \times d_k}, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_v \in \mathbb{R}^{N \times d_v} \tag{5}$$

$$\mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}, \quad \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{N \times N} \tag{6}$$

The attention output is then calculated as follows:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \in \mathbb{R}^{N \times d_v} \tag{7}$$

The query, key and value vectors represent the input in different spaces. The query vector encodes what the model is looking for, like a question that expresses your intrest. The key vector encodes the characteristics of all available information. The value vector represents the actual information you will use. This way the model can learn to focus on the relevant parts of the input and ignore the rest.

Task 3.3.2: Transformers

To find the best architecture, I did a grid search over the parameters `dim`, `depth` and `mlp_dim`, because these are the most important parameters. I kept the number of heads fixed at 8, the batch size at 128 and the number of epochs at 20. The results are shown in Table 3. Comparing the results of the different architectures, one can see that the accuracy is very similar for most of them. The training time however increases with the number of dimensions. The parameter `dim` has the biggest impact on the accuracy. A bigger dimensionality leads to better results. The depth and the MLP dimension have a smaller impact.
The best model had a dimension of 128, a depth of 8 and an MLP dimension of 128. The accuracy on the test set was 0.9871. The training loss shows a lot of variance, but the model was able to learn. The training loss is shown in Figure 14.

Rank	dim	depth	mlp_dim	accuracy	Exec Time
1	128	8	128	0.9871	351.1s
2	128	6	256	0.9863	309.6s
3	128	4	512	0.9860	455.0s
4	256	4	512	0.9859	730.3s
5	128	8	64	0.9857	351.4s
6	64	8	128	0.9856	356.4s
7	64	4	128	0.9851	264.4s
8	64	8	256	0.9849	355.6s
9	64	6	128	0.9847	310.6s
10	32	8	128	0.9845	350.2s
11	128	6	128	0.9842	500.1s
12	128	8	256	0.9841	352.6s
13	256	8	64	0.9839	999.3s
14	128	10	64	0.9838	710.6s
15	128	6	64	0.9837	311.1s
16	128	4	256	0.9837	270.3s
17	32	10	256	0.9835	605.7s
18	64	8	64	0.9835	356.8s
19	64	4	64	0.9834	281.3s
20	128	10	128	0.9833	758.1s
21	32	8	256	0.9833	353.7s
22	128	4	128	0.9833	269.3s
23	64	6	256	0.9831	461.7s
24	256	10	128	0.9831	1302.0s
25	128	4	64	0.9829	268.8s
26	32	8	64	0.9829	351.6s
27	256	8	128	0.9827	1067.3s
28	64	6	64	0.9827	309.7s
29	32	8	512	0.9820	529.6s
30	32	6	128	0.9819	311.4s
31	32	6	64	0.9816	313.3s
32	64	4	256	0.9816	267.8s
33	256	10	256	0.9815	1432.3s
34	32	6	256	0.9809	403.2s
35	32	4	512	0.9808	301.9s
36	32	4	256	0.9808	266.2s
37	64	4	512	0.9807	358.5s
38	32	4	128	0.9798	266.7s
39	32	4	64	0.9788	268.2s
40	32	6	512	0.9778	418.6s

Table 3: Accuracy of different Transformer models on MNIST dataset

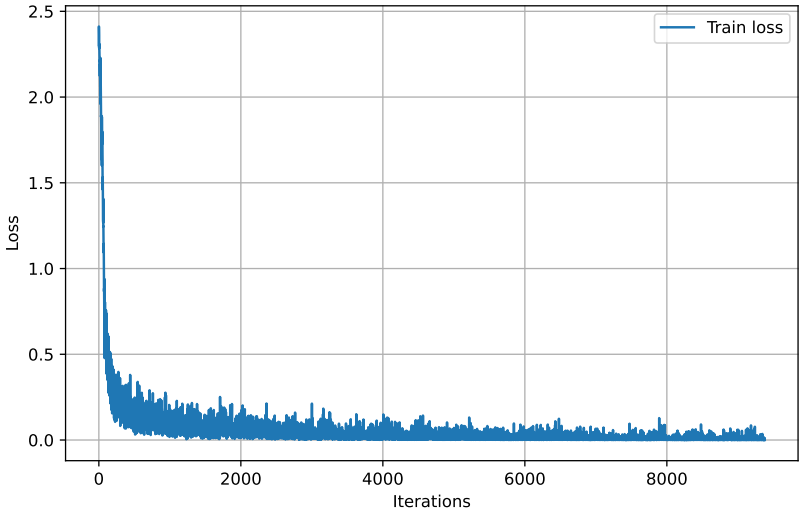


Figure 14: Training loss of the best Transformer model.

4 Generative Models

4.1 Energy-Based Models

Restricted Boltzmann Machines

Task 4.1.1

Pseudo-likelihood is similar to likelihood in the sense that it is a measure of how well the model fits the data. However, it is not a true likelihood function. Instead, it is a product of conditional probabilities of each pixel given all other pixels. This makes it easier to compute and is a good approximation of the true likelihood.

Task 4.1.2

With a low learning rate, the model converges slowly which would need a higher number of iterations to reach the same performance. On the other hand, a high learning rate can cause oscillations in the training process. This can be seen in the variance of the pseudo-likelihood values.
The number of components had the biggest impact on the performance. With too few components, the reconstructed images were blurry and did not resemble the original data. The number of iterations needed to be big enough for the model to converge. Chosing a higher number of iterations did not improve the performance significantly. I assumed the model to be converged based on the pseudo-likelihood values in Figure 15.
The performance of the model was evaluated visually by reconstructing unseen test images. The best results were achieved with 200 components, a learning rate of 0.1, and 30 iterations. Counterintuitively, the model produced more convincing results than other models that achieved a higher pseudo-likelihood. This could be due to the fact that the pseudo-likelihood is not a perfect measure of the model's performance.

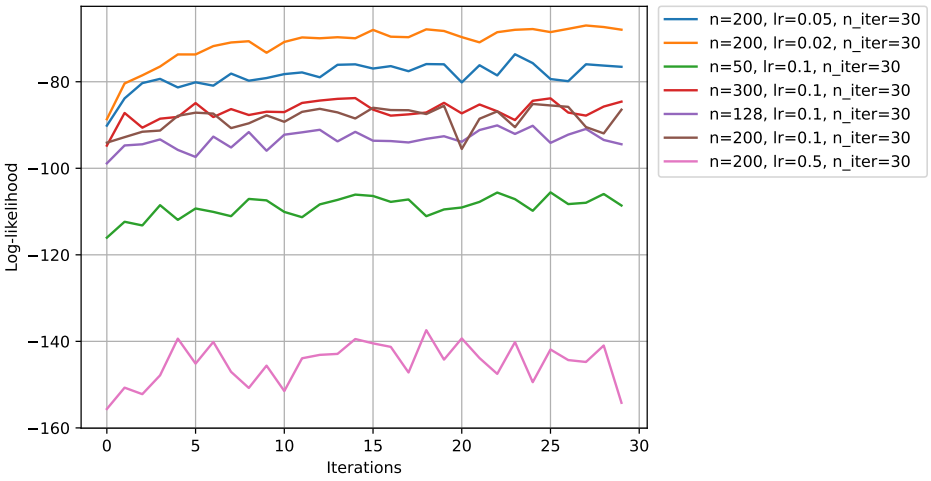


Figure 15: Pseudo-likelihoods of RBM training with different parameters.

Figure 16: Reconstructed images of a 9 with Gibbs sampling steps. Number of Gibbs sampling steps: 1, 2, 5, 10, 100, 1000.

Task 4.1.3

The effect of the number of Gibbs sampling steps can be seen in Figure 16. With only one Gibbs sampling step, the model can already reconstruct the image quite well. With more steps, the image only improves slightly. After more than 10 steps, the image degrades and converges to something that could resemble a different digit or the mean of all digits. This could be related to the problem of iterated functions, where the model converges to a fixed point.

Task 4.1.4

In this example, 12 rows at the top of the image were removed. The RBM is able to reconstruct some digits better than others, as shown in Figure 17. Two of the 0-digits have decent reconstructions. The digit 1 works particularly well, likely because of its simple shape. The digit 9 is falsely reconstructed as a 4. However, the model manages to reconstruct the digit 5 correctly, even though it could also be seen as a 3 with the top part missing. In this case, the model has the most trouble with the second 0-digit, because of the additional stroke in the middle.

Task 4.1.5

When removing more rows, the quality of the reconstruction decreases quickly. With 16 rows removed, the model only manages to reconstruct about half of the digits correctly. Removing rows on the bottom, left or right side of the image has a similar effect. With removed rows in the middle, the model has more trouble, likely because the middle of the image contains more context than the edges.

Figure 17: Top: Original images with missing parts. Middle: Reconstructed images. Bottom: Merge of original and reconstructed images.

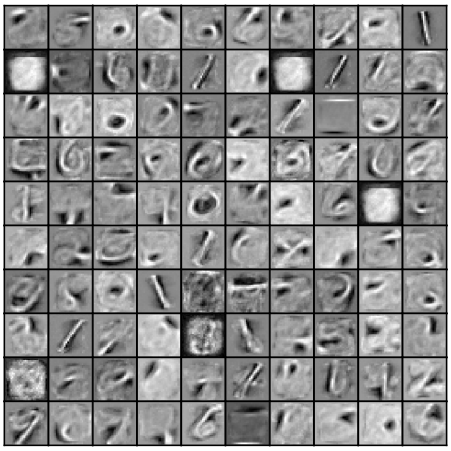
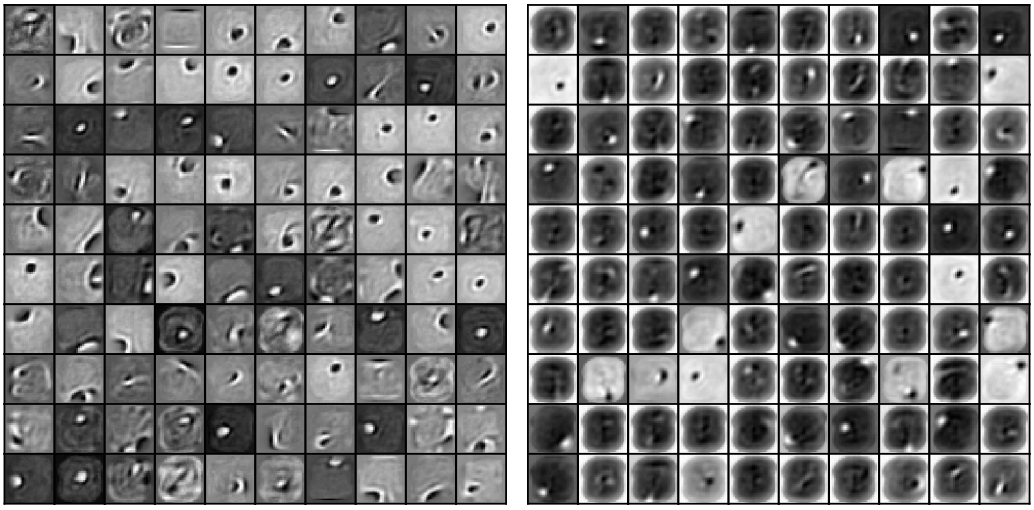


Figure 18: Components of the RBM.

Deep Boltzmann Machines

Task 4.1.6

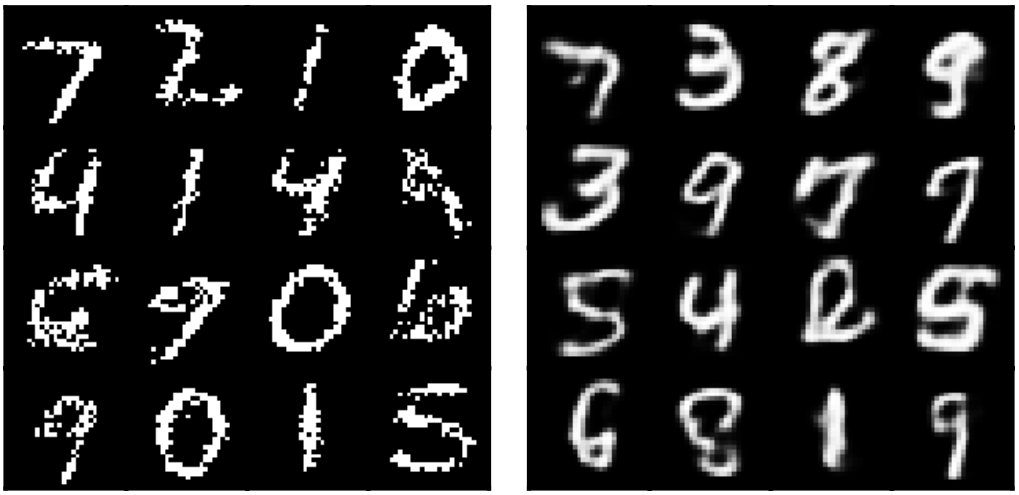


(a) First layer (b) Second layer

Figure 19: Components of the DBM.

Comparing the components of the RBM (see Figure 18) and the first layer of the DBM (see Figure 19a), the components of the RBM have many more lines and structures that resemble the digits. Most of them look unstructured and seem to focus more on the edges, because of dark lines directly next to white lines. The components of the DBM on the other hand often have a dark background with a bright center or vice versa. This could be related to the fact that the DBM has more layers and every component can focus on more abstract features.
The majority of the components of the second layer have white edges and a dark center, as shown in Figure 19b. Many of them also have a spot in the middle, but it is not as pronounced as in the first layer. Overall, the components of the second layer have much more contrast than the first layer.

Task 4.1.7



(a) RBM (b) DBM

Figure 20: Samples generated after 1 Gibbs sampling step.

The samples generated by the RBM and the DBM after 1 Gibbs sampling step are shown in Figure 20. The samples of the DBM are of higher quality than the samples of the RBM, because they resemble digits more closely and have less artifacts. The biggest difference is that the RBM only produces binary pixel values, while the digits of the DBM have continuous edges, like anti-aliasing was applied. Also, the structure of the digits is more consistent in the DBM samples.

4.2 Generator and Discriminator in the Ring

Task 4.2.1

The loss of the generator measures how good the generator is at fooling the discriminator. The generator tries to minimize this loss by generating samples that are similar to the real data. The loss of the discriminator measures how good the discriminator is at distinguishing between real and fake samples. The discriminator tries to minimize this loss by correctly classifying the samples.

Task 4.2.2

If the discriminator performs much better than the generator, the generator will have a hard time fooling the discriminator. This could lead to mode collapse, where the generator only produces a few different samples. The generator will not be able to learn from the feedback of the discriminator.

Task 4.2.3

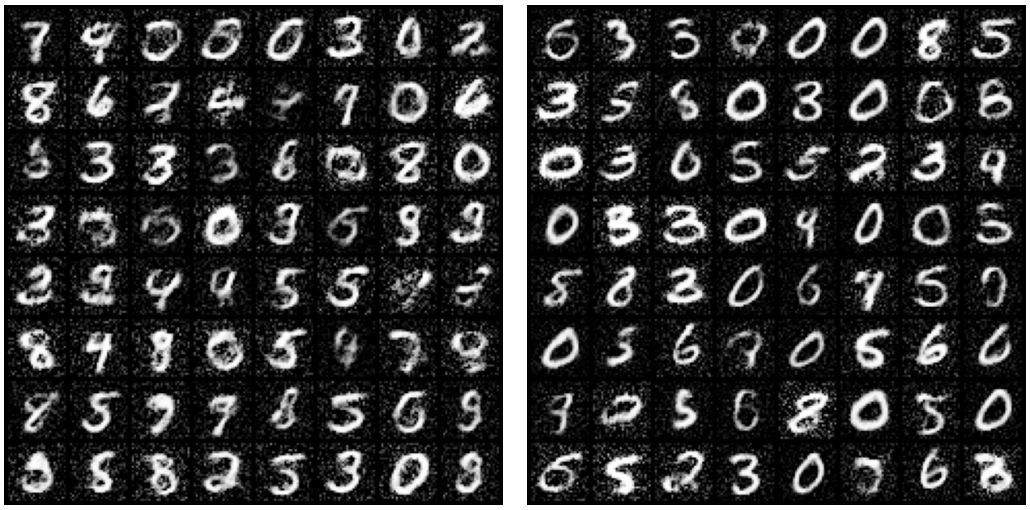
The losses of the generator and discriminator during training are shown in Figure 22. In the optimal case, both losses should converge to a low value, so that the generator and discriminator reach an equilibrium. The losses should be balanced, so that the generator can learn from the feedback of the discriminator. In this case, the losses of the generator increase. However, this does not mean that the generator is getting worse, because the discriminator is also getting better.

If the losses of the generator and discriminator are not balanced, the generator will not be able to learn from the feedback of the discriminator. This could lead to mode collapse, where the generator only produces a few different samples.

Task 4.2.4

A batch of random samples generated by the GAN is shown in Figure 21. Most of the samples resemble the digit 0. The structure of the digits is clear for about half of the samples. The model seems to prefer round shapes, as there are only a few samples that could be interpreted as a 4 or 7 and none that could be interpreted as a 1. Also, all samples have a noisy background, some more than others.

Task 4.2.5



(a) Feedforward backbone (b) CNN backbone

Figure 21: Samples generated by the GAN with different backbones.

A comparison of the samples generated by the GAN with the different backbones is shown in Figure 21. In general, the samples generated by the CNN backbone are of higher quality. The structure of the digits is clearer and the background is less noisy. There are only a few samples that are hardly recognizable as digits. The CNN backbone also prefers round shapes and the digit 0 is also the most common digit.

Task 4.2.6

The main advantage of GANs is that they can generate samples that are very realistic, due to the adversarial training. However, GANs are hard to train and can be unstable. They are also prone to mode collapse, like described above. Auto-encoders are easier to train and can also be used for dimensionality reduction. However, they are not as good at generating realistic samples. Diffusion models are also good at generating realistic samples, but they need longer to train. They are also harder to understand than GANs.

4.3 An Auto-Encoder With a Touch

Task 4.3.1

The model maximizes the evidence lower bound (ELBO) instead of the log-likelihood. The ELBO is the sum of the (negative) reconstruction error and the Kullback-Leibler divergence

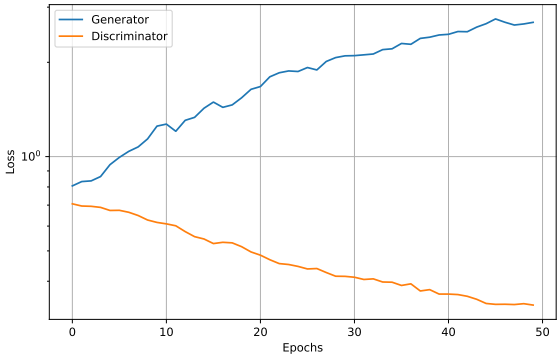


Figure 22: Losses of the generator and discriminator during training.

Figure 23: Random samples from the latent space of the VAE.

between the latent distribution and the prior distribution. The reconstruction error expresses how good the model is at reconstructing the input data. The KL-divergence measures how much information is lost when the latent distribution is approximated by the prior distribution. The ELBO is used because it is easier to optimize than the log-likelihood. Also, the KL-divergence term acts as a regularizer that prevents the model from overfitting.

Task 4.3.2

The main difference between the stacked auto-encoder and the VAE is that the VAE uses a probabilistic approach to model the latent space. This allows the VAE to generate new samples by sampling from the latent space. The stacked auto-encoder uses cross entropy and the VAE uses binary cross entropy as the metric for the reconstruction error.

Task 4.3.3

I trained the VAE with latent-space dimension 300 and middle layer dimension 1024 for 100 epochs. The samples from the latent space are shown in Figure 23. One can tell that they are supposed to be digits, but only a few of them have a clear structure. There are some samples that are not recognizable as digits at all. Others can be seen as a digit, but are not clearly defined. A big majority of the outputs look like the digit 8. Also, there is no sample that is clearly a 0-digit. I consider this as counterintuitive, because of its simple shape.

Task 4.3.4

The VAE generates samples by sampling from the latent space, while the GAN generates samples by feeding random noise through the generator. The VAE has the advantage that it can generate samples from the latent space, while the GAN can only generate samples from the input space. This makes the VAE more interpretable, because you can see how the latent space is structured. The GAN has the advantage that it can generate more realistic samples, because it is trained to fool the discriminator.