Lucas De Vrieze (r0665032)

Comparison of learning algorithms

A neural network was trained using different learning algorithms to approximate the function $y = \sin(x^2)$ for x in a range between 0 and 3π with a step size of 0.05. The network has a single dense hidden layer of 30 neurons with a *tanh* activation function. Each algorithm trained the same 50 randomly initialised networks. The dataset was split randomly: 70 % as the training set, and 15 % as validation and test set each. Training the network was stopped when the MSE did not decrease for six consecutive epochs or when 100 epochs have passed. The following algorithms are being compared:

- Gradient descent (`traingd`, GD)
- Gradient descent with momentum and adaptive learning rate backpropagation (`traingdx`, GDX)
- Conjugate gradient backpropagation with Fletcher-Reeves updates (`traincgf`, CGF)
- BFGS quasi-Newton backpropagation (`trainbfg`, BFG)
- Levenberg-Marquardt backpropagation (`trainlm`, LM)
- Bayesian regularization backpropagation (`trainbr`, BR)

The metrics below were determined for all 50 trials and averaged to compare the performance of the learning algorithms above. This was done at epochs 1, 15 and 100 to monitor the training progress.

- The MSE on the test set
- The regression between the data and the network predictions for the entire dataset
- The average number of epochs required to converge.

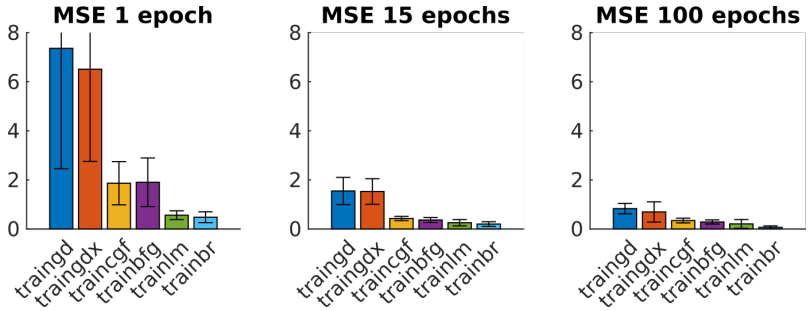


Figure 1: Test MSEs at different epochs.

Figure 1 shows that the two gradient descent methods perform the worst while the LM and BR score the best in all three cases. This is no surprise as LM – and its regularisation derivative BR – allows to interpolate between gradient descent and the Newton method due to its damping parameter λ. Nevertheless, the backlog of GD and GDX and also CGF and BFG diminishes with an increasing number of epochs. The performance of the gradient descent methods appears to be still improving at higher epoch numbers, while those of CGF, BFG and LM seem to have stalled at 100 epochs in comparison with the situation at 15 epochs. In contrast, BR managed to improve even further.

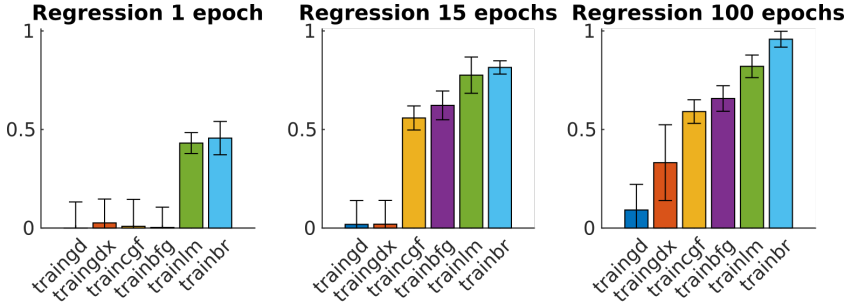


Figure 2: Regression between the data and the network predictions for the entire dataset.

Figure 2 illustrates the increasing alignment between the data and the network predictions. Remarkably, both LM and BR already reach a regression of over 0.4 in the first epoch. Both CGF and BFG quickly catch up after 15 epochs, when both gradient descent methods still reported almost none regression. At the later epochs, the gradient methods managed to capture some trend, with an advantage for GDX due to its momentum and adaptive learning rate. Again, BR managed to differentiate itself from LM and reached a regression of over 0.9.

Table 1: Average required number of epochs to converge.

Algorithm	GD	GDX	CGF	BFG	LM	BR
Average required epochs	95.50	68.76	16.98	17.78	20.02	96.78

Table 1, however, shows a disadvantage of BR: it needs many more epochs to converge compared to LM. Both gradient descent algorithm need many epochs to reach their mediocre results as well.

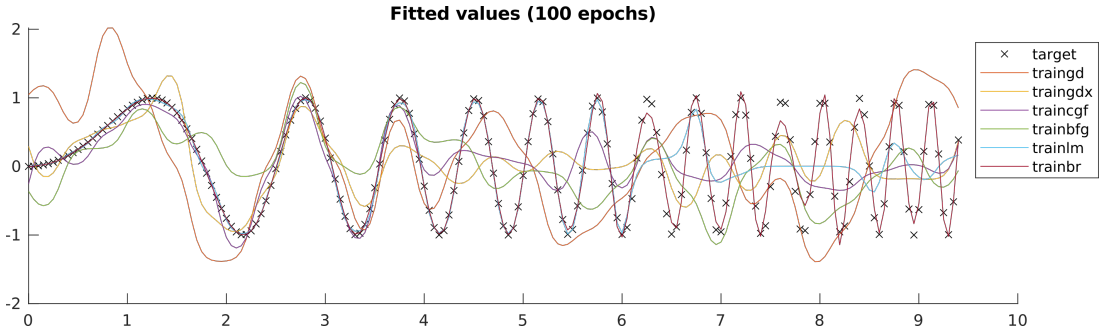


Figure 3: Fitted values for the network with the lowest MSE.

Figure 3 demonstrates the superior performance of BR, which can approximate the function well over the entire x range up to 3π. LM is able to reproduce the function until an x value of about 2π.

Generalisation from noisy data

The procedure of the section above was repeated for the same function, but now with added white noise with a standard deviation of 0.2.

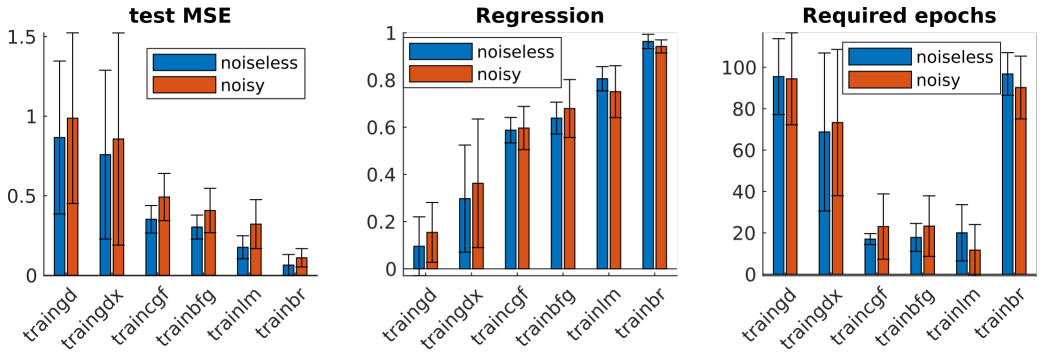


Figure 4: Comparison of the performance of training on noiseless vs. noisy data.

Figure 4 shows the same metrics while training using data with and without noise. As expected, the performance is slightly worse for noisy data as the general trend is not that clear anymore, which reflects in the MSE being slightly higher. There are no clear trends in the average number of required epochs: some algorithms require more, others less. Remarkably, GD, GDX, CGF and BFG manage

to reach slightly more regression when noise is incorporated. Yet, the same general conclusions as the previous section hold.

Bayesian regularisation for overparametrised networks

In theory, Bayesian regularisation favours simpler models that are more easily generalisable and thus reduces overfitting, especially in the case of overparametrised networks. Hence, this reflects in Figure 5, in which BR still scores better than LM regarding test MSE. Interestingly, not much epochs are required to fit overparametrised networks as there are enough neurons to capture any existing trend, both genuine or noisy ones. Moreover, BR trained on noisy data reaches a higher regression than LM trained on clean data. Comparing with Figure 4, one can also notice that the test MSE is much higher for an overparametrised network, indicating serious overfitting on the training data.

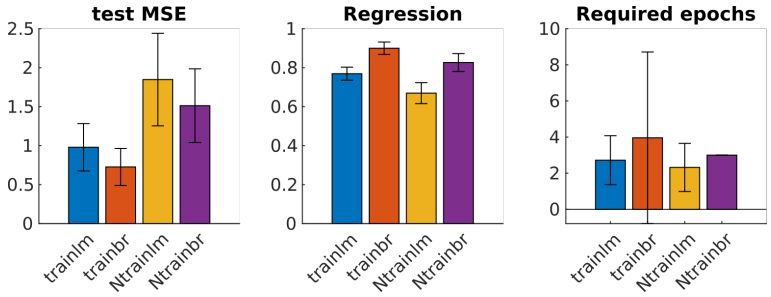
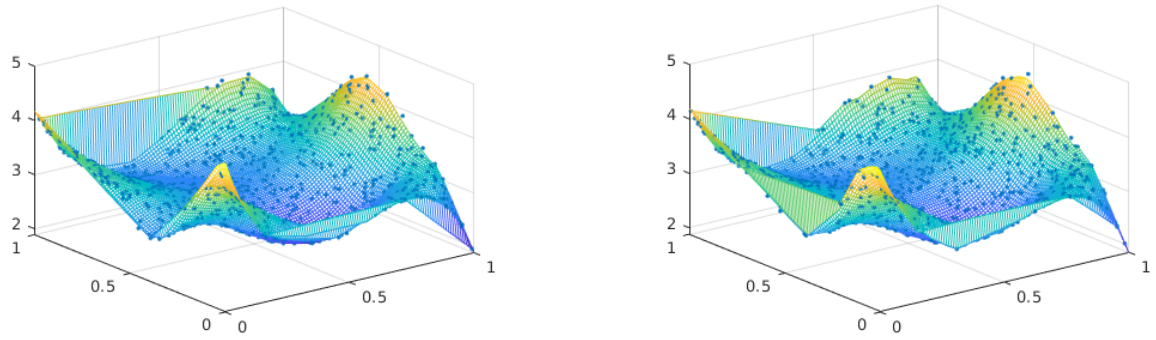


Figure 5: Performance metrics of LM and BR for an overparametrised (300 neurons) network trained on both noiseless and noisy data for 100 epochs. An N before an algorithm name denotes noisy data.

Approximating a 2D surface

Setup of the data subsets

A sample of 3000 points was randomly drawn from this new 2D surface dataset without replacement and split into three subsets serving as the training, the validation and the test dataset. This ensures that there are no shared observations in either of the three subsets. If an observation would be in both the training set and the validation or test set, the performance metrics would likely be biased optimistically as the network generally performs better for a training sample. The training set is used to fit a network, while the validation set serves to determine when to stop training. The test set is employed to determine the network's performance metrics. Figure 6 visualises the training and test subsets' surfaces.



(a) Training set **(b)** Test set

Figure 6: Surfaces of the training and the test sets. Samples within each subset are shown as dots.

Building the network

For the activation function, I hesitated between ReLU and *tanh*. The two primary reasons to choose a ReLU function are that it limits the vanishing gradient problem as the function does not saturate in one direction, and that it is computationally more efficient due to its quasi-linearity. At the other hand, *tanh* may be interesting for this case as it naturally maps inputs to values between -1 and 1, similarly as the 2D surface to be approximated. Moreover, if I would use shallow networks with a limited number of layers, the vanishing gradient problem may be not that much of a concern. So, I decided to use the *tanh* activation function.
I chose to use the Levenberg-Marquardt algorithm as it offers a high accuracy at a fast convergence. To determine the structure of the network, I first fixed the total number of neurons to be distributed over the layers at 60, which is a relatively high number with a convenient number of integer divisors. As such, I picked a number of layers by lowest test MSE averaged over 10 training runs of 200 epochs. Then, I tuned the number of neurons within each layer, again by test MSE averaged over 10 trials of 200 epochs. To reduce the computational efforts, the number of neurons in each layer was increased by steps of 3.

Tables 2 and 3 show that I eventually built a network consisting of four layers of 12 neurons. The approximated test surface is plotted in Figure 7 and shows a good agreement with Figure 6b. Yet, it may be possible to achieve an even better performance by extending the training process. Figure 8 shows that the performance was still improving at 200 epochs. Also note the magnitude difference between the training performance at one hand and the validation and test performance at the other hand. This indicates a certain level of overfitting has taken place, which could be mitigated by applying a Bayesian regularisation algorithm.

Table 2: Average test MSE for an increasing number of hidden layers.

Number of hidden layers	1	2	3	4	5
Average test MSE ($\cdot 10^{-6}$)	36.35	4.51	3.19	**1.35**	1.52

Table 3: Average test MSE for an increasing number of neurons within each of the four layers.

Number of neurons	6	9	12	15	18	21	24	27	30
Average test MSE ($\cdot 10^{-7}$)	370.15	24.29	**8.65**	10.38	25.17	33.95	28.70	45.63	60.89

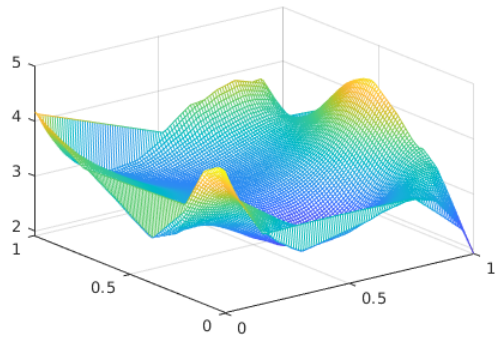


Figure 7: Surface of the test set as approximated by the final network

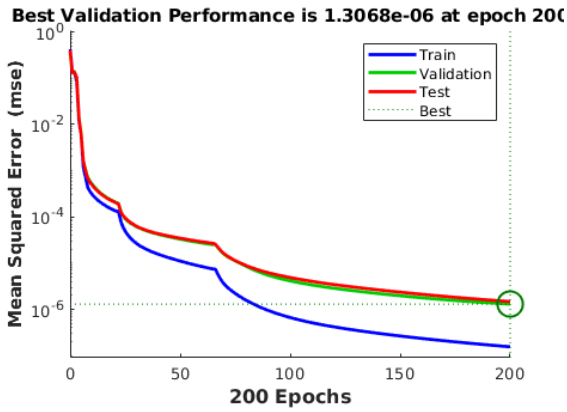


Figure 8: Training performance progress plot of the final network

Lucas De Vrieze (r0665032)

Hopfield networks

Hopfield networks serve as a model for associative memory. After feeding it an input, the network dynamically evolves to an equilibrium state, an attractor. One can design a Hopfield network so that it has some desired attractors, although additional undesired attractor states cannot be excluded. Here, the convergence and attractors of Hopfield networks with two and three neurons are examined.

Two-neuron network

A two-neuron Hopfield network was instantiated with explicit attractor states $(1, 1)$, $(-1, -1)$ and $(1, -1)$. After generating 100 random initial vectors with values between -1 and 1 and feeding each one to the network as an initial state, the dynamical behaviour of the network was simulated for 50 time steps. This showed that there is another unwanted attractor state in $(-1, 1)$. However, this is not a surprise as for every attractor ξ that is explicitly coded, its negative $-\xi$ is stored as well.
The dynamical trajectories of these random initial points in Figure 1 allowed to identify possible saddle points. By initialising the network at a point on the imaginary dividing line between the trajectories of the random initialisation points, four saddle points $(0, 1)$, $(0, -1)$, $(-1, 0)$ and $(1, 0)$ were identified, as illustrated in Figure 1. Moreover, the origin emerged as unstable attractor state.
Figure 2 shows the number of iterations to reach the attractor state for all simulations, both the random and non-trivial points at the dividing line. Most simulations converge within 20 iterations. Points that take longer, appear to be random ones that are close to a dividing line.

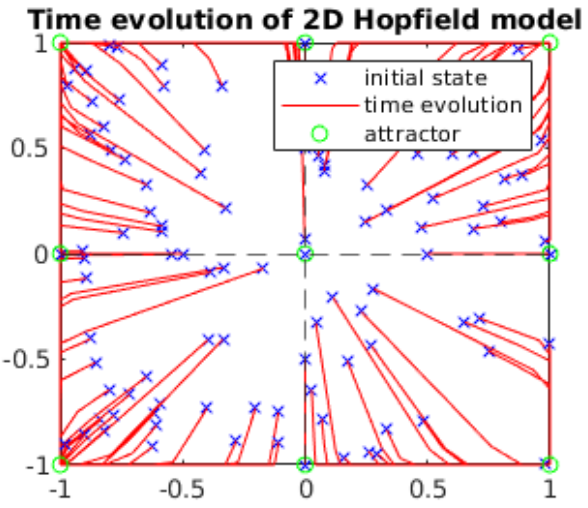


Figure 1: Trajectories of the evolution of the two-neuron network initialised at 100 random points and some points on the dashed dividing lines.

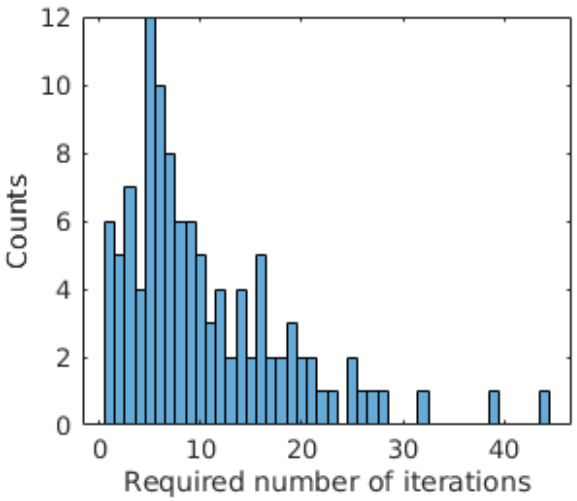


Figure 2: Histogram of the number of required iterations to reach the attractor state for the two-neuron network.

Three-neuron network

Similarly, a three-neuron network was instantiated with explicit attractor states $(1, 1, 1)$, $(-1, -1, 1)$ and $(1, 1, -1)$ and simulated for 500 time steps using 100 random initial vectors. The three explicit attractor states appear to be the only ones in this case, as illustrated in Figure 3. Again, from the dynamical trajectories, there appear to be dividing curves between the attractor states in the 3D space. However, in this case, it is not that easy to pinpoint these curves. Initially, I would guess the dividing curve is simply the perpendicular bisector of two attractor states. Yet, all trajectories seem to converge towards a curve that is slightly offset, forming a kind of rounded triangle in 3D space.
Figure 4 shows that, again, most trajectories converge within 20 iterations, although some points closer to the dividing curves take it longer.

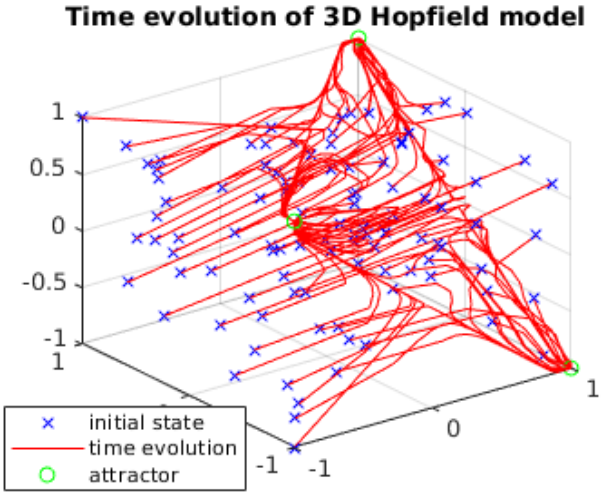


Figure 3: Trajectories of the evolution of the three-neuron network initialised at 100 random points.

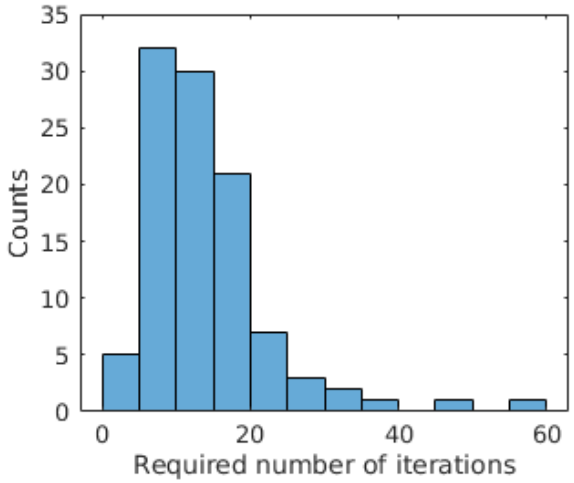


Figure 4: Histogram of the number of required iterations to reach the attractor state for the three-neuron network.

Hopfield networks for hand-written digits

This subsection tests the capacity of a Hopfield network to correctly classify hand-written digits by evolving to the correct attractor digit upon receiving an arbitrary digit as the input state. To do so, its classification strength was tested for an increasing amount of noise added to the original image and an increasing amount of network iterations. It was observed that the higher the noise, the less digits were correctly classified. Especially for noise levels above 4, performance dropped quickly. Allowing more network iterations regains some classification performance, but the network still struggles with distinguishing digits that have similar shapes, e.g. when both numbers have circles and straight lines. Examples are shown in Figures 5 and 6.

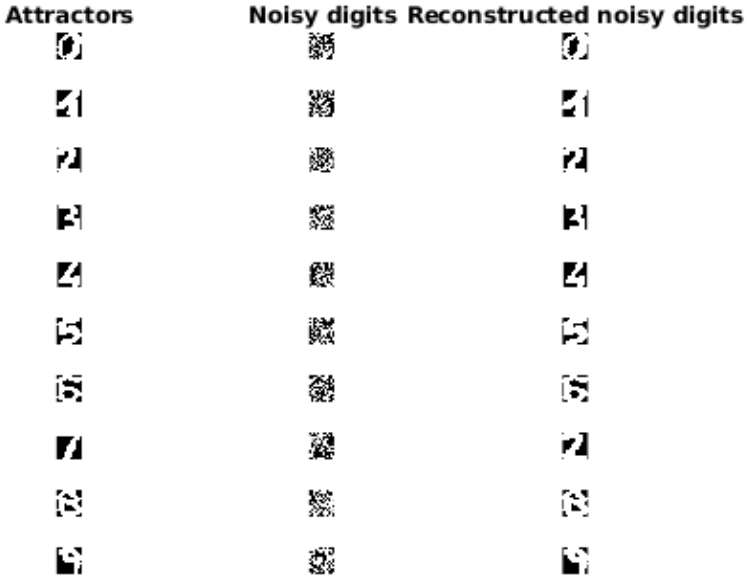


Figure 5: An example of digit images reconstructed after adding white noise (level 6) and 10,000 iterations of the Hopfield digit network. The 7 has been wrongly classified as the similar 2.

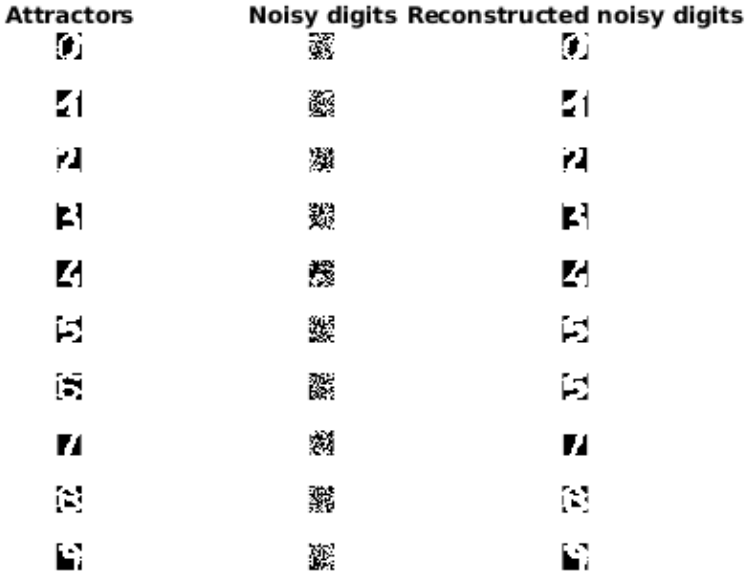


Figure 6: An example of digit images reconstructed after adding white noise (level 6) and 100,000 iterations of the Hopfield digit network. The 6 has been wrongly classified as the similar 5.

Time-series prediction

Using multi-layer perceptrons (MLPs)

The MLP network consists of only one hidden layer. The size of the lag l was varied between 5 and 50 with steps of 5, while the number of neurons N in the hidden layer was varied between 20 and 80 with steps of 5. For each combination of parameters, 10 randomly initialised networks were trained for 250 epochs using Levenberg-Marquardt. The training time-series data was split like before: 70 % training, and 15 % each for validation and test set. Early stopping was applied if the validation performance did not improve for six consecutive epochs. Next, the first newly predicted datapoint was inferred from the lag. The new prediction was added to the lag vector in FIFO-style, like a moving data horizon. Then, a new value was predicted from the lag vector, starting a new prediction cycle.

The performance metric at hand is the prediction RMSE for the test time-series dataset, averaged over the 10 trials. Figure 7 shows a heatmap of this average RMSE for all examined parameter combinations (l, N). The prediction performance improves with increasing lag size, especially for lags larger than 15. The number of neurons does not appear to have an impact for larger lag sizes.
Next, the parameter combination with the lowest average RMSE was selected (lag size of 35, 50 neurons). Then, the network with the lowest RMSE for that parameter combination was chosen to produce the final prediction, which is shown in Figure 8. It appears that the MLP network is able to reproduce the first high-amplitude part of the test set quite well, but fails to predict the low-amplitude part after discrete time point 60.

Using long short-term memory networks (LSTMs)

The LSTM network consists of a sequence input layer that represents a lag of size l, a LSTM layer with a number of hidden units N, a fully connected layer and a regression layer. It was trained using ADAM for 250 epochs. The same moving data horizon approach was adopted to predict the test time-series data. Tuning happened similarly as before: by picking the parameter combination (l, N) from a heatmap with the lowest prediction RMSE averaged over 10 trial runs (here: lag size of 30, 60 units), and then the network with the lowest RMSE of the 10 trials for that combination.

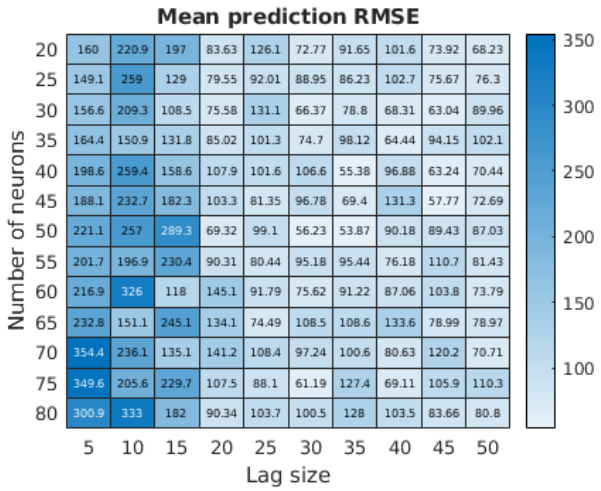


Figure 7: Mean prediction RMSE heatmap for all examined parameter combinations of the MLP network, averaged over 10 trials.

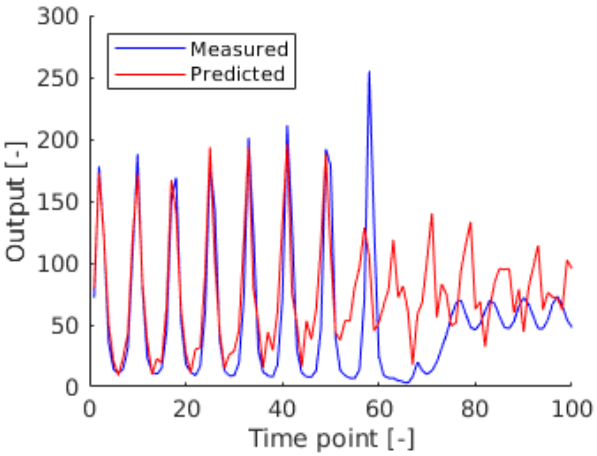


Figure 8: Predicted time-series by the final MLP network versus the original test data.

From the heatmap in Figure 9, another trend than for the MLP is visible. The observation that the RMSE decreases with increasing lag size still holds, but the number of hidden units now appears to have a modestly positive influence on the RMSE instead of none. Note that the RMSEs of the LSTM networks are substantially lower than those of the MLP networks as well.
Figure 10 shows the prediction by the selected LSTM network. It is clear that it does a better prediction job than the selected MLP network. It almost perfectly aligns with the first high-amplitude part, while it scales down its amplitude in the next low-amplitude part. However, there is a phase shift in the predictions for that part of the test data.

To conclude, I would choose LSTM networks over the MLP option because of the superior prediction performance they demonstrated for this case.

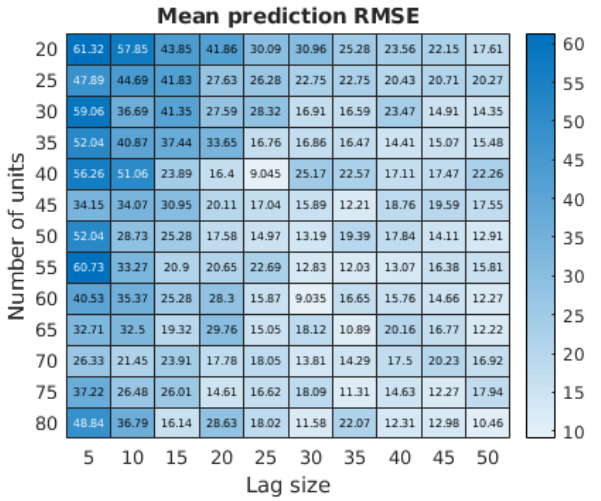


Figure 9: Mean prediction RMSE heatmap for all examined parameter combinations of the LSTM network, averaged over 10 trials.

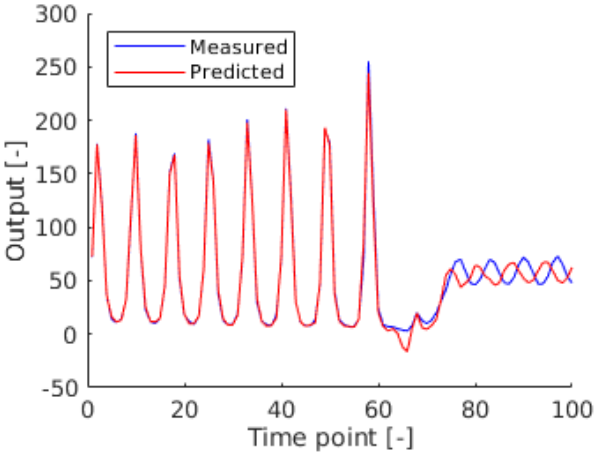


Figure 10: Predicted time-series by the final LSTM network versus the original test data.

Lucas De Vrieze (r0665032)

Principal component analysis

A custom PCA function was built using Matlab's built-in functions and applied on a couple of datasets.

PCA on random vs. correlated data

A PCA of random n-dimensional data should return an entirely different result than one on correlated data. PCA maps the dataset to a lower-, p-dimensional space, trying to preserve as much information as possible, which is related to the data variance in a particular dimension. At one hand, a random dataset should not contain much correlation, as each original dimension is independently sampled from the same Gaussian distribution. At the other hand, in a correlated dataset, some dimensions are very similar and could be replaced by one linear combination of those original dimensions.
Reconstructing the original n-dimensional data from PCA-reduced data and recording the reconstruction RMSE gives insight in how much information was contained in its reduced p-dimensional representation. Figure 1 shows that, for the correlated dataset, the reconstruction error levels off to zero for reduced spaces with more than 10 dimensions. This implies that most information in this dataset can be expressed in 10 dimensions, effectively removing half of the original dimensions (21). For the random data, all dimensions are required to get a good reconstruction. This shows that there indeed was no correlation in the random dataset.

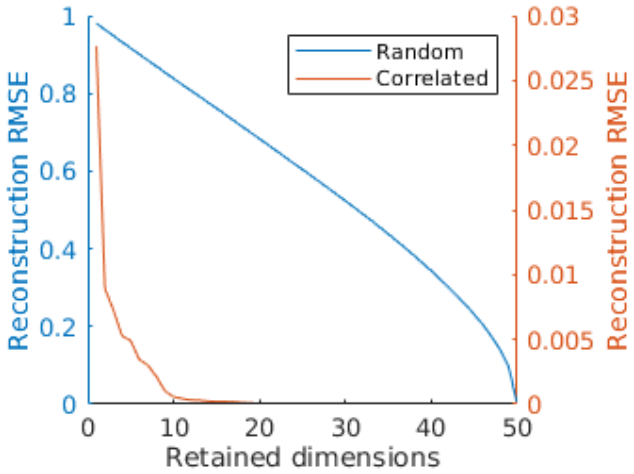


Figure 1: Reconstruction RMSE for both the random and the correlated dataset.

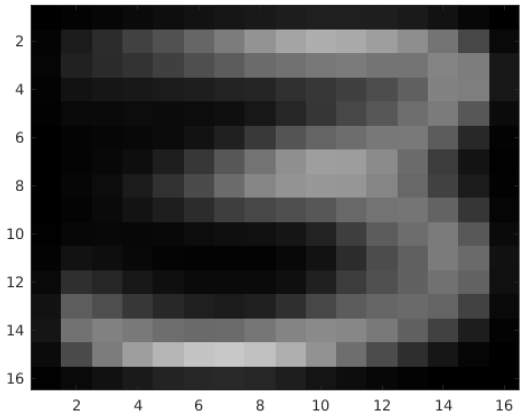


Figure 2: Mean image of a 3 digit.

PCA on hand-written digits

The mean 3 is depicted in Figure 2. Figure 3 shows the reconstructions of one image after dimensionality reduction to one up to four dimensions. It is clear that the more dimensions in the compressed image, the better is the reconstruction. This observation is confirmed and quantified in blue in Figure 4 after compressing all images in the provided dataset into p dimensions with p between 1 and 256. Remarkably, the RMSE for 256 dimensions – the dimension of the original dataset – is not zero, albeit very small: $7.22 \cdot 10^{-16}$, which contradicts intuition. A possible reason for this is that, although it does not actually compress the image, a PCA still rotates and scales the axes of the 256-dimensional space so that its dimensions are ordered by the amount of variation in each dimension, i.e. by eigenvalue. As such, it may incorporate a very small numerical error due to the finite machine precision.
Figure 4 also illustrates another observation. The larger the sum of the eigenvalues of the p first dimensions, the smaller the reconstruction RMSE. This aligns with the reasoning that the sum of the eigenvalues relates to the variance and thus the amount of information that the p dimensions capture.

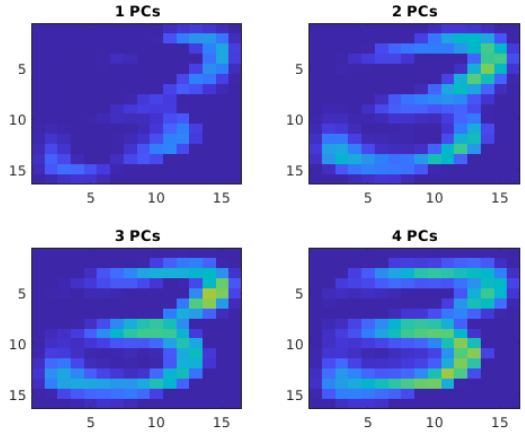


Figure 3: Reconstructed image for one 3 digit from one to four compressed dimensions

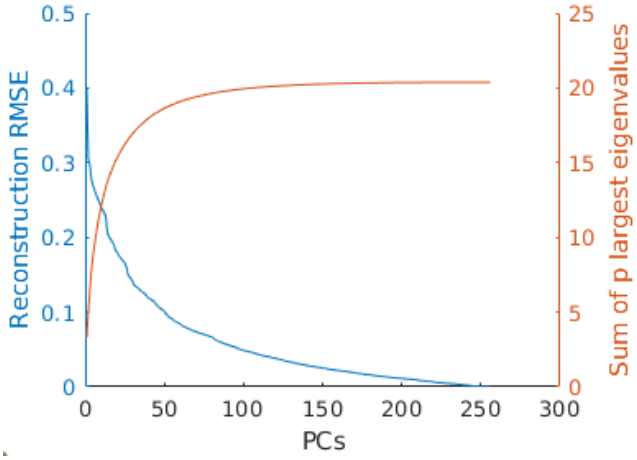


Figure 4: Reconstruction RMSE for and sum of the p largest eigenvalues.

Stacked autoencoders

This section compares the performance of stacked autoencoders to the one of normal multilayer neural networks. The initial stacked network in the provided script contains two stacked autoencoder layers and a softmax layer. The first autoencoder contains 100 hidden units and was trained for 400 epochs, while the second one has 50 hidden units and was trained for 100 epochs. The softmax layer was trained for maximally 400 epochs. Fine-tuning lasted until the gradient dropped below 10^{-6}.
Several alterations were made to this architecture to optimise the number of hidden units, the number of epochs and the number of layers. Additionally, two normal neural networks (one with one layer of 100 hidden units and one with two layers of resp. 100 and 50 hidden units) were trained for 200 epochs or until the validation performance did not improve for six consecutive epochs (architectures 1L and 2L resp. in Table 1).

Initially, I observed that the performance of the first autoencoder in the initial architecture (nr. 1 in Table 1) levels off at about 200 epochs. So, there is not really a point in continuing training. At the other hand, the second autoencoder, appears to be still improving at 100 epochs, so I extended its training process with another 100 epochs (cfr. architectures 2 and 3). Then, I changed the number of units within the layers. First, I examined the impact of doubling the number of units in both layers versus only in the first layer (cfr. architectures 4 and 5). I also assessed the case of halving the number of units in the second layer (architecture 6). The last architectures alter the number of layers and assess which of the layers that I already defined in the earlier cases, perform better on their own (cfr. architectures 7 to 11). I randomly initialised 10 networks for each architecture and averaged their accuracies. The results of the normal multilayer networks and of the stacked networks before and after fine-tuning are given in Table 1. It appears that only one fine-tuned autoencoder layer can already outperform a multi-layer network.

Fine-tuning a stacked network involves retraining the entire network after its constituent parts have been trained individually. It applies back-propagation through the entire network to tweak the weights with respect to the full process at once. It has been shown that this practice decreases the required training time compared to training the full network from scratch. It avoids the vanishing gradient problem and increases the likelihood of finding a good minimum during the gradient descent.
Interestingly, fine-tuning increases the performance of all networks up to a similar test accuracy level of about 99% accuracy. Remarkably, the architectures that did perform the best before fine-tuning end up being the worst after fine-tuning, and vice versa.

Table 1: Test accuracy of the examined network architectures before and after fine-tuning, if applicable, averaged over 10 randomly initialised instances of each architecture. 'Number of units' is the number of units in each network layer. 'Number of epochs' is the number of epochs a certain layer has trained individually.

	Number of units	Number of epochs	Mean test accuracy before fine-tuning	Mean test accuracy after fine-tuning
1	100/50	400/100	85.08	99.73
2	100/50	200/100	87.65	99.76
3	100/50	200/200	94.84	99.25
4	200/100	200/200	97.97	98.81
5	200/50	200/200	95.07	99.13
6	100/25	200/200	65.04	99.72
7	200/100/50	200/200/200	62.22	99.74
8	100	200	98.50	98.94
9	200	200	99.50	99.62
10	50	200	91.92	99.18
11	100	400	98.50	98.97
1L	100	max. 200	97.14	NA
2L	100/50	max. 200	96.81	NA

Convolutional neural networks

AlexNet

This section takes a look at the AlexNet CNN. The first convolutional layer is located in layer 2, directly after the image input layer. Its 96 weights have dimensions 11×11×3 and represent the 96 convolution kernels of this layer, each representing one particular image feature.
As ReLU and Cross-Channel Normalisation layers do not alter the dimensions of their input, it is possible to determine the dimensions of the input to layer 6. The key formula for this goal has been derived by Dumoulin and Visin for square matrices via reasoning about the number of possible placements of the kernel matrix on the input matrix [1]. It is given below, with o the output dimension, i the input dimension, k the kernel dimension, p the size of the padding and s the stride size.

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1$$

Given that $\lfloor (227 + 2 \cdot 0 - 11)/4 \rfloor + 1 = 55$ and that there are 96 convolution kernels, the output of layer 2 is of dimension 55×55×96. For a max pooling layer like in layer 5, the same formula can be applied, omitting the padding term [1]. Hence, with $\lfloor (55 - 3)/2 \rfloor + 1 = 27$ for 96 matrices, the input of layer 6 is a block with dimensions 27×27×96.
Assuming that grouped convolutions alter the dimensions of the data in the same way as one large convolution, applying the same formula leads to final inputs of dimensions 6×6×256. Comparing this to the original dimensions of 227×227×3, a down-sampling ratio of about 17 has been achieved.

The preference of CNNs over fully connected networks for image classification roots in its primary characteristics. Firstly, a convolutional network is **translation-invariant**. Sharing the parameters of the neurons (weights and bias) over the entire filter does not only benefit the computational tractability, it also makes the neurons respond to the same feature and output a feature map as response to a certain input image. Due to their identical behaviour, it does not matter where a particular feature is situated, as any neuron of the convolutional layer can detect it. Moreover, the max pooling layers

contribute to this translation invariance as well. When outputting the maximum value in their visual field, they neglect the position of this maximum value.
Secondly, a convolution kernel only examines local features due to its **local connectivity**, whereas a fully connected layer takes all inputs into account, also the far removed ones. Again, this local connectivity does not only lower the computational efforts, it also enables CNNs to detect very local features, like a line with a certain inclination, or a bright dot, in a smaller patch of the full input image. Although this makes a convolution kernel responsive to only one small spatial feature, stacking pooling and other convolutional layers allows to combine the responses of different kernels into a more complex image, similarly as happens in the brain.

Trying out some CNN architectures on hand-written digits

I started from the original architecture (nr. 1 in Table 2) in the `CNNDigits.m` script and adapted its parameters (size of the convolution kernels, number of kernels, size of the max pool, stride of the max pool, number of layers) one by one. Networks were trained using the `sgdm` (stochastic gradient descend with momentum) algorithm for 30 epochs and an initial learning rate of 0.0001. The test accuracies in this table show that it is possible to get a better accuracy than the original architecture.

Firstly, implementing more filters increases the accuracy (cfr. architecture 1, 2 and 3), probably because it allows to select for more features.
Secondly, decreasing the filter size increases the accuracy as well (cfr. architectures 1, 4 and 5), maybe because it forces the convolutional layer to look for a limited number of basic local features.
Thirdly, increasing the max pool size (cfr. architecture 5 and 6) reduces the resolution of the signal, perhaps reducing the network's capability to discern larger and smaller features.
Fourthly, increasing the stride in the max pooling layer (cfr. architecture 5 and 7) may generate a weaker signal for a particular feature in the pooled image, because the pooling window shifts faster.
Fifthly, less layers appear to perform at least equally well in this case. Additionally, integrating more filters in less layers appears to increase the accuracy here (cfr. architectures 1, 8 and 9).
Lastly, more layers may work as well, although this raises new questions as to how many filters in each layer and what about the size and the stride of the max pooling layers in-between.

Table 2: Test accuracy of the examined CNN architectures. 'Kernel sizes' are the size of the convolution kernels in each network layer. 'Number of kernels' is the number of kernels in each network layer.

	Kernel sizes	Number of kernels	Max pool size	Max pool stride	Accuracy
1	5/5	12/24	2	2	94.24
2	5/5	10/20	2	2	88.76
3	5/5	15/30	2	2	96.48
4	9/9	12/24	2	2	50.12
5	3/3	12/24	2	2	96.56
6	3/3	12/24	3	2	96.20
7	3/3	12/24	2	3	88.56
8	5	12	NA	NA	94.72
9	5	24	NA	NA	97.12
10	5/5/5	12/24/48	2/3	2/1	95.68

References

[1] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

Lucas De Vrieze (r0665032)

Restricted Boltzmann machines (RBMs)

This section examines the impact of changing the parameters of the training process of RBMs. Intuitively, increasing the number of epochs, the learning rate and/or the number of hidden units should propel the performance of the RBM, as is visible in Figure 1. And, indeed, we can see this difference for any configuration that leads to an increased training performance, for example by comparing the images in Figure 2. An RBM with 100 hidden units appears to capture the digits' characteristics better, resulting in clearer images.

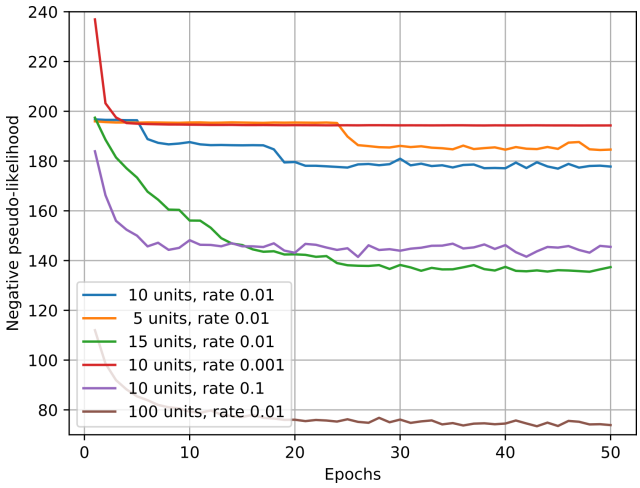
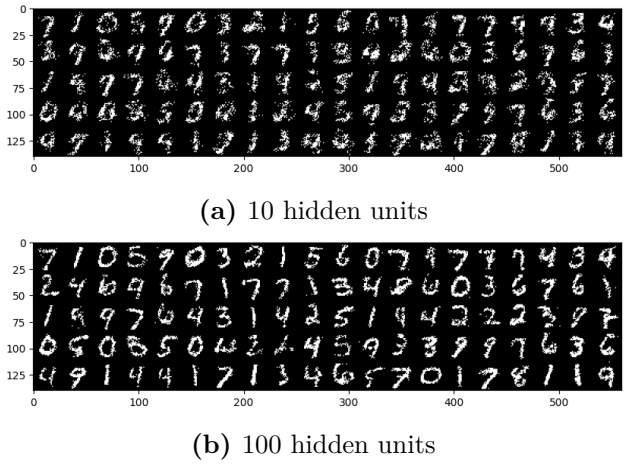


Figure 1: Training performance plot of the RBM for different training parameters.

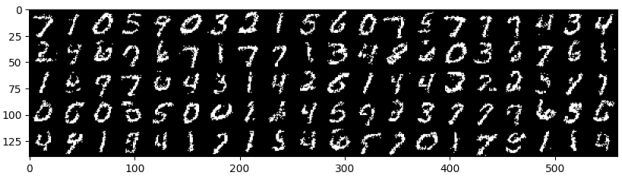


(a) 10 hidden units

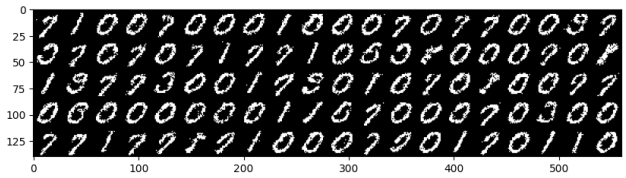
(b) 100 hidden units

Figure 2: Gibbs-sampled (1 step) images from an RBM with 10 hidden units vs. one with 100.

A remarkable parameter is the number of Gibbs sampling steps. The general trend is the more sampling iterations, the more images evolve into the same few apparent digits, which I will call 'attractor digits', as illustrated in Figure 3. This may be explained by reasoning about how Gibbs sampling works. An inherent aspect is updating conditional probability distributions with a newly drawn value for one of the variables using a Monte Carlo algorithm. We are sampling an RBM here, so the probabilities can be interpreted in this case as the likelihood that one pixel takes on a certain value given the values of all other pixels in the image, as modelled by the RBM from its training dataset. So, chances are higher that the sampler updates a certain pixel with a new value that is correlated with the other pixel values. Hence, after a large number of sampling cycles, the most common shapes will arise due to their higher pixel correlation and assemble into the few apparent attractor digits.



(a) 10 Gibbs sampling steps

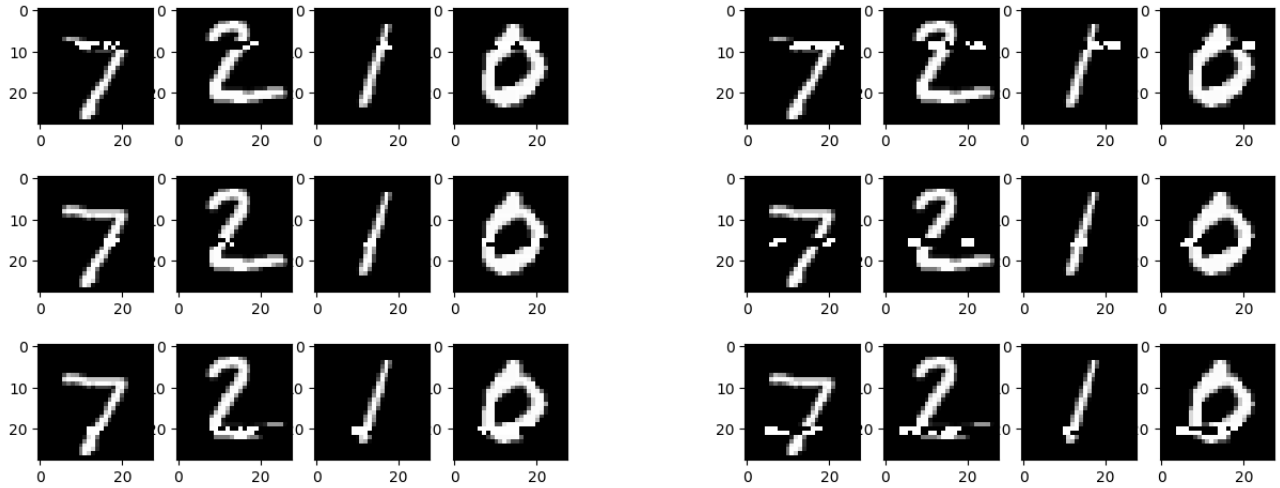


(b) 1000 Gibbs sampling steps

Figure 3: Gibbs-sampled images from an RBM with 100 hidden units.

Regarding image reconstruction, a well-trained RBM is able to reconstruct small patches of digit images. As a result, RBMs with more hidden units, more training epochs and/or higher learning rates perform better. Regarding the number of Gibbs sampling steps, the sweet spot seems to be between 1 and 10 sampling steps, as illustrated in Figure 4. It probably preserves the general shape of the original digit before letting it slide into an attractor digit. Regarding the location of the reconstructed

rows, the ones in the centre appear slightly better than rows in the top or the bottom (see Figure 4a), possibly because digits have more diverse shapes at their extremities than in their centre.



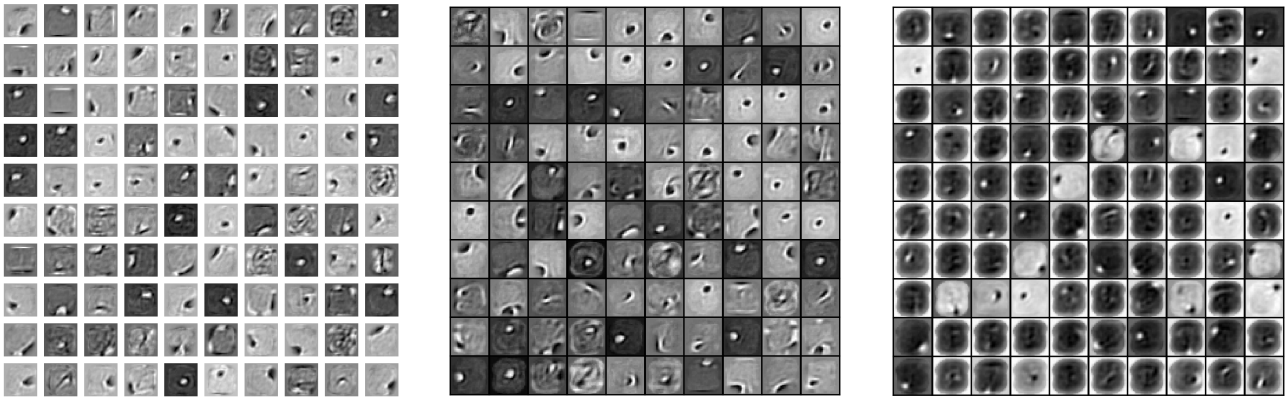
(a) 10 Gibbs sampling steps **(b)** 10,000 Gibbs sampling steps

Figure 4: Digit images with reconstructed rows at a different location and for a different number of Gibbs sampling steps.

Deep Boltzmann machines (DBMs)

The first layer of a DBM and the RBM layer, both depicted in Figure 5, do not appear very different in my opinion. Both select for the presence (light dots) or absence (black dots) of certain regions. Important to note here is that these are weight plots for an RBM with 100 hidden units. Intuitively, the weight pictures of the layer of an RBM with 10 hidden units would look more blurry and noisy.

The second layer of the DBM does differ from the first layer. Whereas the first layer filtered for the features themselves and appears overall grayish, the second layer returns a more black-and-white picture. I would interpret this as that the second layer is sorting combinations of present and absent features as identified by the first layer, to link them to a certain digit.
Sample images taken from the DBM are clearly less noisy than those from the RBM. Hence, the DBM appears to capture more complex features and probability distributions using its stacked architecture.



(a) RBM layer **(b)** Layer 1 of the DBM **(c)** Layer 2 of the DBM

Figure 5: Comparison of the weights of the first 100 components of the RBM and the DBM layers

Lucas De Vrieze (r0665032)

Generative adversarial networks (GANs)

The deep convolutional GAN (DCGAN) for class 1 (cars) of the CIFAR dataset was trained for 20,000 iterations. The DCGAN appears inherently unstable in Figure 6, as the loss and the accuracy of both the generator and the discriminator fluctuate between 0.7 and 0.8, and 0.35 and 0.75 respectively. Training a GAN is a dynamical process in which two models compete in a game-like setting. When one model improves its accuracy, it usually comes at the cost of a lower accuracy for the other model. The goal of this process is that this dynamical behaviour eventually converges in an equilibrium state. Yet, often, it ends up in an oscillation between different states, as might be the case here.

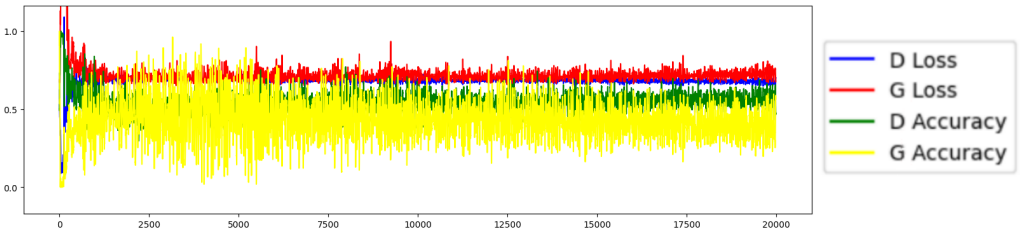


Figure 6: Accuracy and loss of both the discriminator and the generator models of a DCGAN trained for 20,000 iterations on the cars class of the CIFAR dataset.

Colour swapping

The provided notebook carried out optimal colour transport for the two images included in the zip file for this part. These original images and the results for both the EMD and the Sinkhorn distance metrics are depicted in Figure 7. We can clearly see that the colour palettes have been transported between the two images. For example, the blueish sky patches of image 2 have been transferred to the clearer sky patches of image 1, while the white of the central clouds of image 1 have been transported to the translucent sun in image 2. Comparing the EMD and the Sinkhorn results, it is noticeable that the Sinkhorn images are more homogeneous due to the entropic regularisation term.

If we would just swap the pixels and transfer the colour palette without caring for the shortest distance between the pixels, I think the image would degenerate to nonsense, in which the earlier contrasts and shapes have disappeared. The reasoning behind this is that shapes usually have similar colours. So, colours with the shortest distance to the colours of a certain shape are very similar to each other as well, giving rise to the same well-demarcated shape in the colour palette of the transformed image.

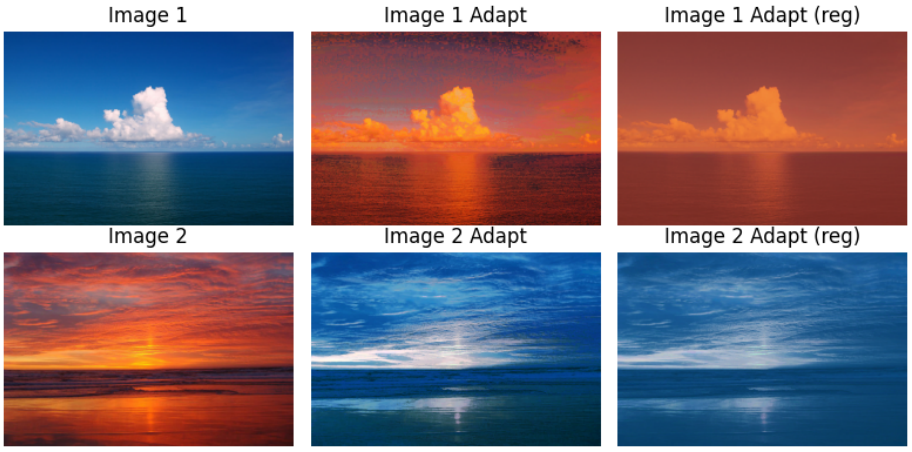


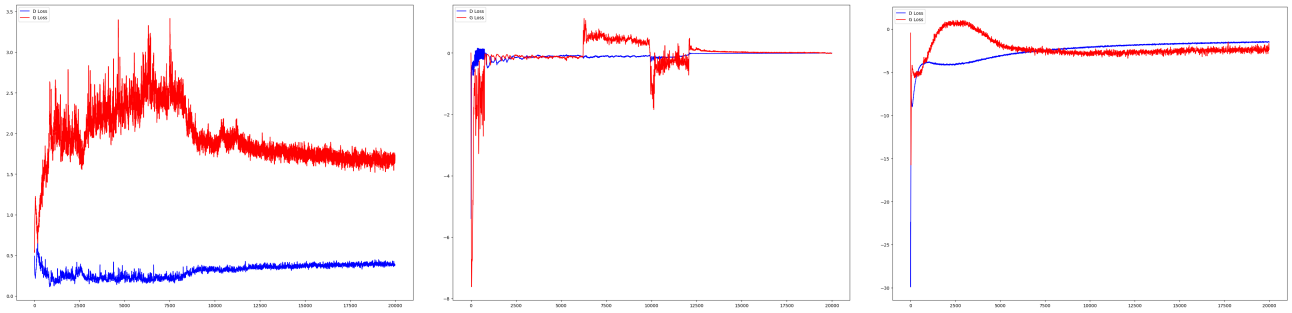
Figure 7: Original images and results from EMD (Adapt) and Sinkhorn (Adapt reg) colour swapping.

Wasserstein GANs

This subsection compares the training process of a standard GAN with two Wasserstein GAN types, one with weight clipping and one with a gradient penalty. The evolution of the loss terms with the number of training cycles is depicted in Figure 8. The standard GAN was, again, unstable and its loss fluctuated relatively much for both models. The WGAN with weight clipping did reach an equilibrium point at a loss around 0 for both the discriminator and the generator. The loss of the WGAN with gradient penalty seems to level off at -2.5 at 20,000 epochs. Yet, there still is some fluctuation in the generator loss.

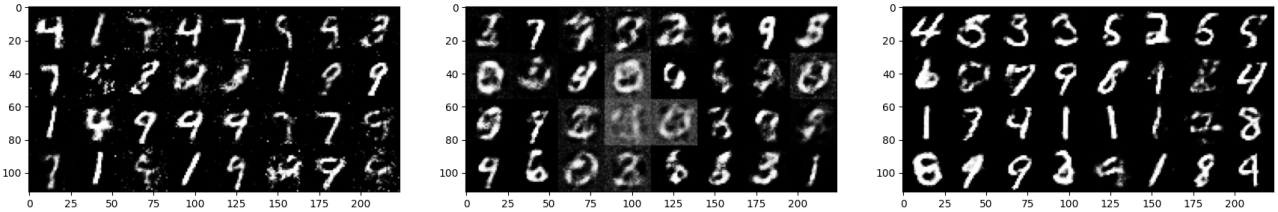
Images generated with the standard GAN are of a lower quality than those of the WGANs. They are, for example, more grainy, as illustrated in Figure 9. Comparing the two WGANs, images generated by the gradient descent WGAN look more clear than those generated by the weight clipping type. Moreover, the weight clipping WGAN sometimes produces a blurred image that does not look like any digit. This might show that reaching an equilibrium point does not necessarily imply a better quality.

The Wasserstein distance is a distance metric between two probability distributions and requires finding the optimal way of transforming one distribution into another, in analogy with transferring colour palettes in images using optimal transport. The discriminator of a WGAN minimises the Wasserstein distance between the probability distribution of the pixel values in the real digits, and the distribution according to the images generated by the generator model. As such, it provides feedback to the generator model about how well it is performing. Moreover, as WGANs omit the log-sigmoid function of standard GANs, the signals do not get saturated at very high divergences, like for low-quality images generated at the start of a WGAN training process. As a result, WGANs may make good learning progress using gradient descent even at the start of the training process.



(a) standard GAN **(b)** WGAN with weight clipping **(c)** WGAN with gradient penalty

Figure 8: Comparison of the loss during the training process of a standard GAN, a WGAN with weight clipping and a WGAN with gradient penalty.



(a) standard GAN **(b)** WGAN with weight clipping **(c)** WGAN with gradient penalty

Figure 9: Comparison of digit images generated by a standard GAN, a WGAN with weight clipping and a WGAN with gradient penalty, all trained for 20,000 epochs.