Artificial Neural Networks and Deep Learning: Report

Thibeau Wouters

June 2023

Contents

All code, data and figures created for this assignment can be found here.

1 Supervised learning and generalization

In this first assignment, we familiarize ourselves with the basics of training neural networks and tuning the architecture design.

1.1 Training algorithms and function approximation

The first example we consider is a simple regression problem where data is sampled from the function $y(x) = \sin(x^2)$ in the interval $[0, 3\pi]$ with spacing of $\Delta x = 0.01$. The test set consists of 100 random samples. The targets are either the true labels or labels with modified targets, obtained by adding Gaussian noise with a factor of $\sigma = 0.2$. We train neural networks with a single hidden layer and investigate training algorithms for varying number of hidden neurons. The training algorithms we investigate are gradient descent (GD), gradient descent with momentum and adaptive learning rate (GDX), Levenberg-Marquardt (LM), conjugate gradient with Fletcher-Reeves (CGF), BFGS quasi-Newton (BFG) and Bayesian regularization (BR).

1.1.1 Training algorithms

First, we train a network with 50 hidden neurons for 500 iterations. The results, averaged over 20 repetitions, are shown in Figure 1.1. As expected, GD is the fastest algorithm, but has poor performance, measured by the mean-squared-error (MSE), since it only uses information from the gradient and has a fixed learning rate. GDX only offers a slight improvement. CGF is slightly slower than both GD and GDX but offers an improved performance. This method also uses gradient information, but additionally makes use of conjugate gradients, which improve the search. The slowest method is the BFG method, and its performance only provides a small improvement over CGF. This is a quasi-Newton algorithm which also has to keep track of an approximation of the Hessian matrix, which can become costly for large networks. Among the slower, but definitely the best performing methods, are LM and BR. The LM algorithm uses the true Hessian matrix (possibly with small perturbations on the diagonal to ensure the inverse exists) and hence uses more information of the surrounding neighbourhood compared to other algorithms. The speed of BR is comparable to LM, since BR builts on LM by adding a regularization term to the loss function used by LM [1]. The BR algorithm searches for the ideal combination of the cost and regularization terms such that the network generalizes well, which results in an improved performance. While the performance depends on the architecture and the dataset under consideration, the LM and BR algorithms seem to overall provide an ideal trade-off between speed and accuracy.

1.1.2 Generalization in function approximation

Next, we consider the generalization properties of the networks, when trained on both noiseless and noisy targets denoted by y and $\tilde{y}$, respectively. For this, we plot the train and test MSE as a function of the number of hidden neurons, averaged over 10 repetitions. We do not rely on a validation set

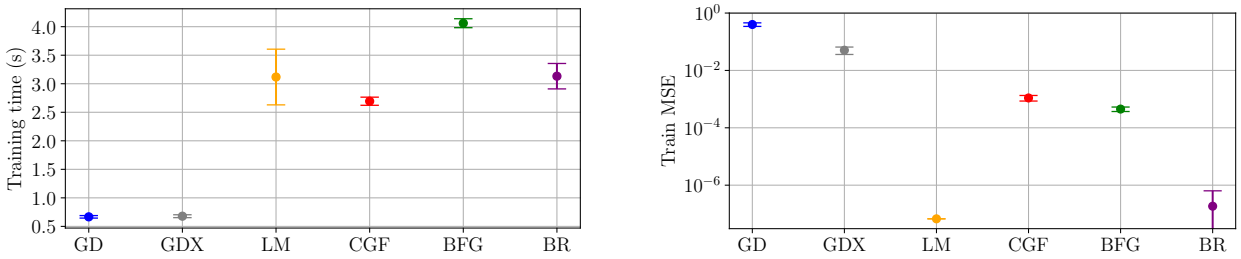

Figure 1.1: Comparison of training algorithms. *Left*: Training time. *Right*: Train MSE.

and hence do not yet use early stopping. The results are shown in Figure 1.2 and Figure 1.3. The networks generalize well for noiseless data and the errors on the training and test sets are comparable to each other. Here, we notice that the GD, CGF and BFG methods' performance stagnates quickly for increasing number of hidden neurons. Especially the GD algorithms are unable to make any noticeable progress as a function of the network size. It is a bit less intuitive why the CGF and BFG methods also quickly stagnate in performance. Likely, the size of the network becomes too large and the loss function too complicated such that these methods are not capable enough to find deep local minima using only gradient information.

The LM and BR methods have the best performance across all network sizes. While both have a comparable performance for architectures up to 70 hidden neurons, the train MSE obtained with LM seems to increase above 80 hidden neurons, likely signalling that the network has become overparametrized. The BR method, due to the additional regularization, does not suffer from this issue and can still ensure an optimal performance of the network.

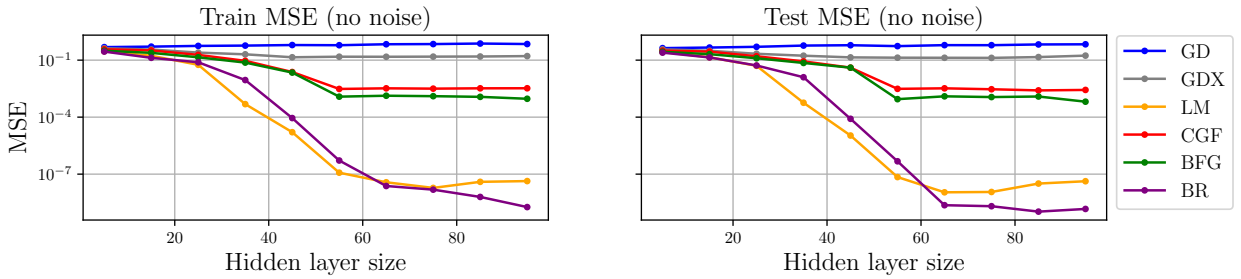

Figure 1.2: Performance of algorithms for varying network size when training on the true targets.

When noise is added to the data, the networks have a higher MSE. As before, the GD methods give the worst results while LM and BR are overall the best methods with CGF and BFG in between the two extremes. Curiously, the MSE on the test sets are lower than on the train sets, although this is likely by chance. The test MSE, measured on $\tilde{y}$, indicates that the best generalization is achieved with $40 - 50$ hidden neurons. When networks are overparametrized (with respect to this optimal size), their test MSE starts to increase while the train MSE stagnates. For very large architectures, all methods (excluding GD and GDX) have a similar performance on the test set, possibly due to overfitting. Hence, the generalization of a network depends on the size and complexity of the network as well as the quality of the data.

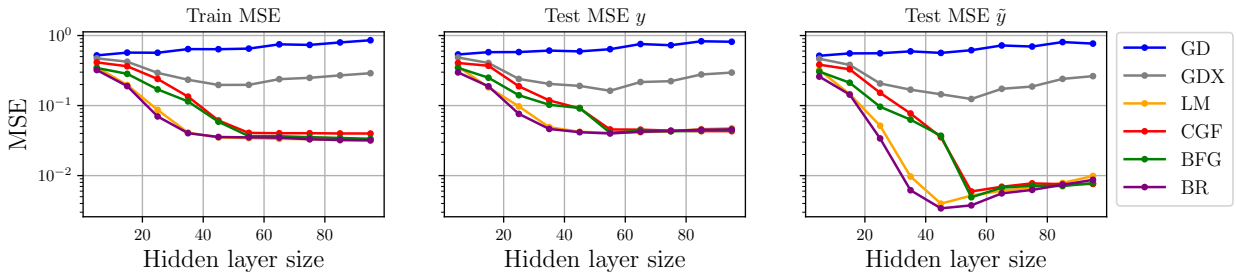

Figure 1.3: Performance of algorithms for varying network size when training on noisy targets.

Finally, we show the results of a network with 45 hidden neurons trained with `trainlm` in Figure 1.4. The network perfectly matches the shape of the function when trained on noiseless data. When trained on noisy data, overfitting seems to mainly originate from the region $x \leq 4$, perhaps due to the slower oscillations.

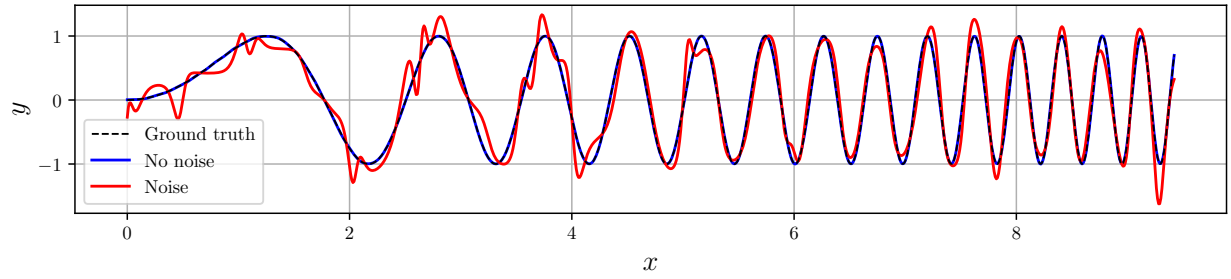

Figure 1.4: Performance of networks with 45 hidden neurons trained with `trainlm`.

1.2 Personal regression problem

Now, we approximate the unknown function built from the given datasets.[1] We create a training, validation and test set in order to properly design our architecture. The training set is used to train the network and adjust its parameters (weights and biases) to improve its performance by minimizing the loss function. The validation set monitors the performance on a separate dataset during training. This allows us to detect whether the network is overfitting and to perform early stopping if needed. Moreover, we can compare validation errors between architectures with different hyperparameters in order to tune the model, *i.e.* select the hyperparameters with the best performance. Finally, the test set is yet another independent set (that is, containing unseen data) used to check the generalization error of the final model selected after tuning on the validation set is completed.

Tuning a neural network can be done through a grid search as follows. We determine for each hyperparameter a range to be searched, take evenly spaced points from these ranges and create a

[1] In particular, we have $d_1 = 8, d_2 = 8, d_3 = 7, d_4 = 5, d_5 = 1$.

grid of all combinations of these points. Each combination is trained and tested on the validation set. Here, we limit ourselves to tuning the architecture of the network and fix the activation functions of the hidden layers to hyperbolic tangents and use the LM algorithm. We restrict ourselves to architectures with up to three hidden layers having an equal number of hidden neurons with a maximum of 30. Each architecture is trained five times for a maximum of 1000 iterations and using early stopping with 10 validation checks before termination. We report the average MSE values, measured on the validation set, in Figure 1.5. Based on our results, we choose an architecture with three hidden layers each containing 10 neurons.

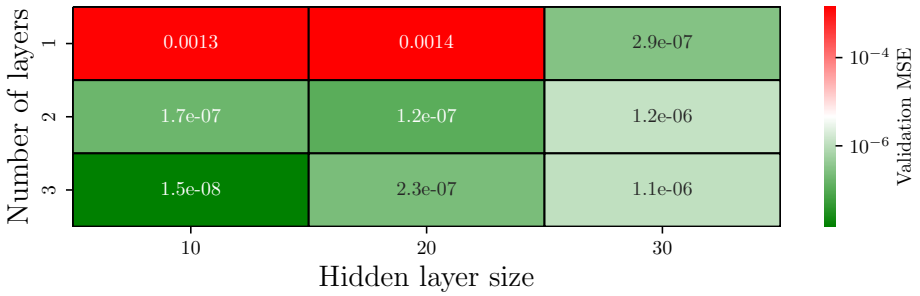

Figure 1.5: Grid search to tune the architecture of the personal regression problem.

After tuning, we train the chosen architecture with the BR algorithm as our earlier observations demonstrated that BR has improved generalization compared to the LM algorithm due to the regularization term. Training a new network from scratch, we obtain an MSE of 2.50×10^{-9} on the training set, 4.41×10^{-9} on the validation set and 3.05×10^{-9} on the test set. The results of the final architecture are shown in Figure 1.6. When plotting the errors on the test set, it is clear that the majority of the error comes from a small region around the origin, which is likely harder to approximate due to the large peak and the small input values. Improvements could be made by weighing the dataset such that datapoints close to the origin have a higher impact on the loss function to improve the training. Moreover, we could use crossvalidation to obtain unbiased and more reliable estimators for the performance of the networks, at the price of increased computational cost.

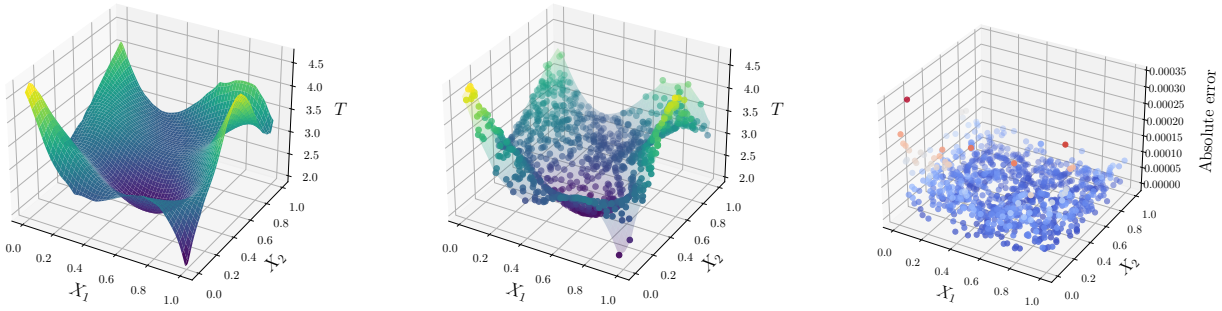

Figure 1.6: Personal regression problem. *Left*: Surface of the training dataset. *Middle*: Surface of the test dataset and predictions of the neural network (dots). *Right*: Absolute errors between predictions and true values of the test set.

2 Recurrent neural networks

In the second assignment, we will delve deeper into recurrent architectures, such as Hopfield networks, recurrent neural networks (RNN) and long short-term memory networks (LSTM).

2.1 Hopfield networks

We start by considering various examples of Hopfield networks. The first Hopfield network is two-dimensional and created by learning three patterns: $(1, 1)$, $(-1, -1)$ and $(1, -1)$. We obtain trajectories of this network for several initial conditions and show them in Figure 2.1. We find 9 different attractors in this system, which include 6 spurious states. Spurious states may arise due to symmetries in the problem formulation. When two or more stored patterns are related to each other by a symmetry transformation, such as a reflection or a rotation, the network can get stuck in a state that is a linear combination of these patterns. This is clearly the case for this problem, since the three patterns are located symmetrically at the corners of a square. In fact, the symmetry is reflected by the learned dynamical system, which is given by:

$$\boldsymbol{x}_{t+1} = \texttt{satlins}\left[\text{diag}\left(1.1618, 1.1618\right) \cdot \boldsymbol{x}_t\right] , \tag{2.1}$$

which has a vector field pointing radially outwards from the origin. Due to the nature of the `satlins` activation function, the attractors at the corners are stable. The origin is an unstable attractor, while the other attractors on the axes are saddle points. It takes on average 8 iterations for the shown initial conditions to converge to an attractor.

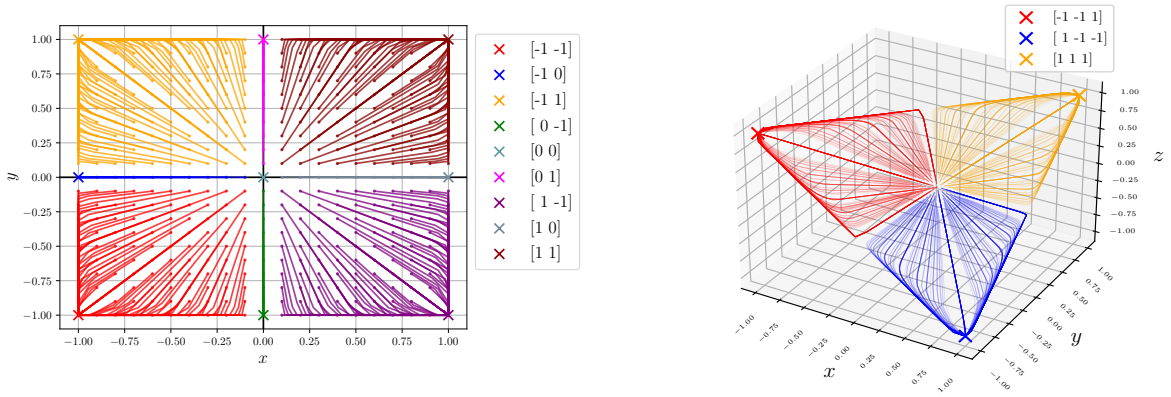

Figure 2.1: Left: Trajectories in the 2D Hopfield network. *Right*: Trajectories in the 3D Hopfield network. Crosses denote attractors found experimentally. Colours denote the attractor at which the trajectory ends.

Next, we consider the 3D Hopfield network shown in Figure 2.1. We found no additional attractors outside of the stored ones. The dynamical system is given by

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t) = \texttt{satlins}\left(\begin{bmatrix} 0.8489 & 0.3129 & -0.3129 \\ 0.3129 & 0.8489 & 0.3129 \\ -0.3129 & 0.3129 & 0.8489 \end{bmatrix} \cdot \boldsymbol{x}_t + \begin{bmatrix} 0.2849 \\ -0.2849 \\ 0.2849 \end{bmatrix}\right) , \tag{2.2}$$

which has no spurious states due to the presence of a bias term which spoils the symmetry. We check explicitly that the other corners are no fixed points of this equation by verifying that $f(\boldsymbol{x}^*) = \boldsymbol{x}^*$ does not hold for these points. On average, it took around 55 iterations to reach the attractors.

As a final example, we consider the Hopfield network for handwritten digit recognition with varying levels of noise added to the digits and number of iterations of the network. We show the number of digits that were correctly reconstructed by the network in Figure 2.2. As expected, low levels of noise are easily restored by the network, while higher levels of noise can potentially disrupt the initial condition and cause the network to converge to the wrong attractor. Generally speaking, a higher number of iterations can improve the performance of the predictions, since higher noise values might give initial conditions farther away from the true attractor such that more iterations are required for convergence. High levels of noise can cause the initial state to lie closer to other attractors which might look similar to the original digits, hence converging towards the wrong attractor in the end.

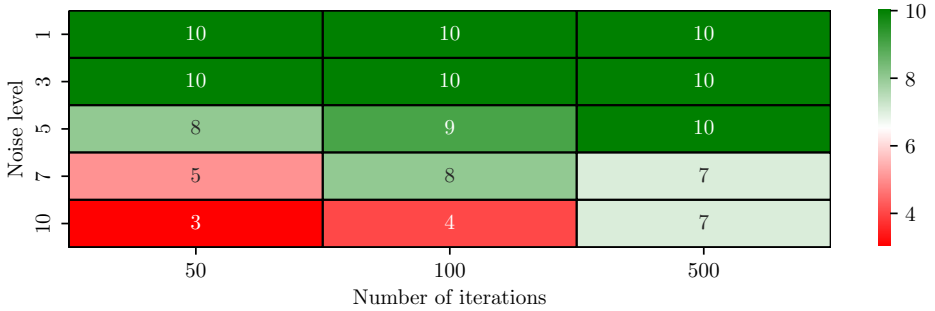

Figure 2.2: Number of correctly classified digits for varying levels of noise and number of iterations.

2.2 Forecasting with RNNs

Next, we consider RNNs for the problem of time series prediction on the Santa Fe dataset. As before, increasing the number of hidden neurons increases the capacity of the neural network but can lead to overfitting. The lag parameter is more important in the architecture design. In a sense, the lag parameter determines the size of the "memory" of the network. A higher lag implies that more (and hence, older) datapoints are taken into account when predicting future values. As a consequence, a low lag value is unable to capture the intricacies of the time series, as we can see from Figure 2.4. Therefore, we want the lag parameter to be as high as possible, although high lag values can make it computationally infeasible to train the network.

One issue is that training the network and testing the network deal with different goals. We train the model as if designed for a regression problem while the network is actually applied in an auto-regressive manner. Since an auto-regressive model has to use its own predictions to predict subsequent values, it ideally has to be capable to handle slight deviations from the true values to ensure a stable time series. Therefore, we use two tuning procedures. The first one is completely analogous to the tuning of a regressor and compares errors on a validation set, in this case the final

30% of the training data. The second tuning method mimics the eventual goal of the RNN: we use the trained RNN as an auto-regressor to predict the time series in the window $[t_{544}, t_{644}]$ of the train set, shown in Figure 2.3, which has a similar shape as the test set.[2] The validation error is then the MSE on this time series, using the RNN as an auto-regressor. The two tuning procedures

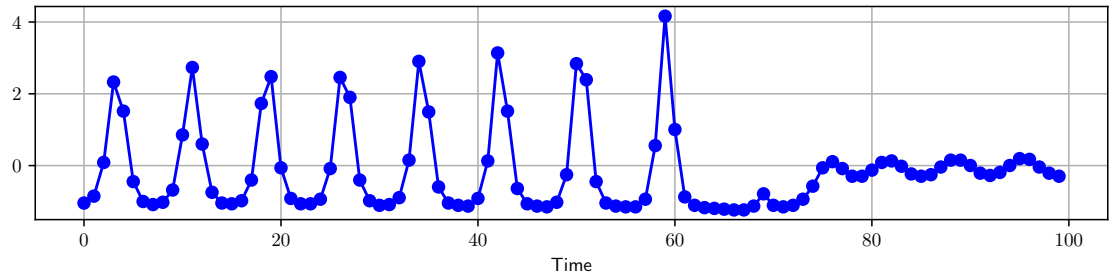

Figure 2.3: Validation set used for tuning the RNN based on its prediction of a time series.

are performed over a grid of (h, p) values, where h is the size of the hidden layer and p is the lag. Both h and p take values from $\{5, 10, 20, 30\}$ and each grid configuration is trained 5 times. The first tuning procedure selects $h = p = 20$, while the second one selects $h = p = 30$.

The results of RNNs predicting the Santa Fe time series is shown in Figure 2.4. We notice that high lag values approximate the first peaks of the test data more accurately. However, all RNNs seem to be unable to adequately capture the sudden drop occurring after 60 timesteps. However, further increasing the lag makes training computationally infeasible.

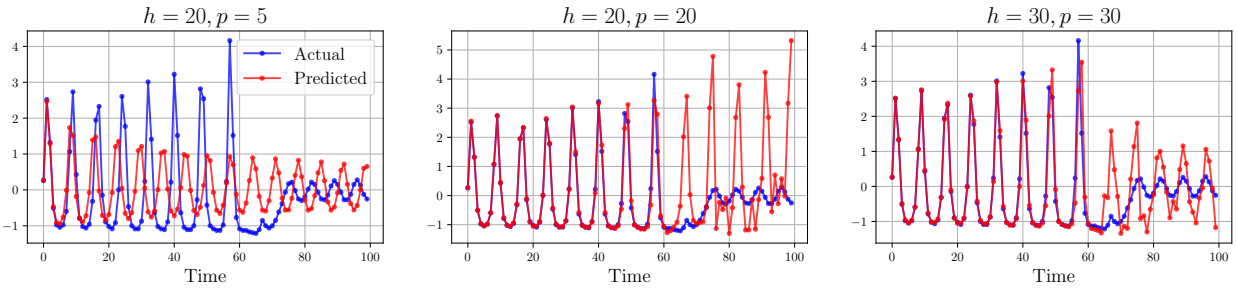

Figure 2.4: Time series forecasting with RNNs. *Left*: Small lag values are unable to capture the sudden drop. *Middle*: Hyperparameters selected by regular tuning. *Right*: Hyperparameters selected by tuning on a time series.

2.3 Forecasting with LSTMs

RNNs are unable to correctly capture chaotic time series and are unstable in the sense that they can easily create divergent predictions when applied on a time series. This can be ascribed to the

[2] We did not manage to properly separate this validation time series from the training data, such that our implementation currently suffers from data leakage. However, we merely provide the second tuning procedure as proof of concept.

fact that RNNs are trained as a regressor, while its purpose is to be applied in an auto-regressive manner.

LSTMs aim to solve this problem by adapting the architecture to the goal of time series prediction. An LSTM uses so-called blocks instead of neurons [2]. Each block has three gates: the input, output and forget gate. The input gate determines which values from the input are used to update memory states. The forget gate determines which information should be deleted from the block. The output gate, finally, determines the output, based on the input as well as the memory of the block. After an LSTM layer, we have a fully connected layer to turn the LSTM output into the desired output for the regression. Due to this particular design, LSTMs excel in modelling long-term dependencies by selectively retaining or discarding information over time. As such, they are more suitable for chaotic data such as the Santa Fe dataset. However, LSTMs are more computationally demanding and require more hyperparameter tuning than RNNs.

We modify the LSTM implementation provided by Matlab for the Santa Fe dataset [3]. Here, we have a single `lstmLayer` of which we can tune the number of hidden units. The optimizer used is the Adam optimizer. Other tunable hyperparameters are the initial learning rate and its adaptation schedule. The current implementation uses a piecewise scheduler, meaning that the learning rate is multiplied with a certain factor (smaller than 1) after a fixed amount of training epochs. Similar to RNNs, the lag value can be tuned, with larger lag values capturing longer-term dependencies in the data.

We modify the provided implementation such that the lag parameter can be tuned. To have a fair comparison with the RNN architectures discussed above, we train an LSTM network with 30 hidden units and a lag value of 30, which were the values that were chosen by our tuning procedure. We train for 300 epochs with an initial learning rate of 0.005 and multiply the learning rate with 0.2 after 150 epochs. The results are shown in Figure 2.5. We note that the predictions are a much better approximation of the true time series. In particular, the LSTM is able to capture the sudden drop after 60 iterations. The predictions are also accurate in terms of MSE values, and only the phase of the oscillations of the final part of the test data is wrong. Hence, LSTMs are more powerful architectures compared to RNN due to their use of memory gates instead of standard neurons.

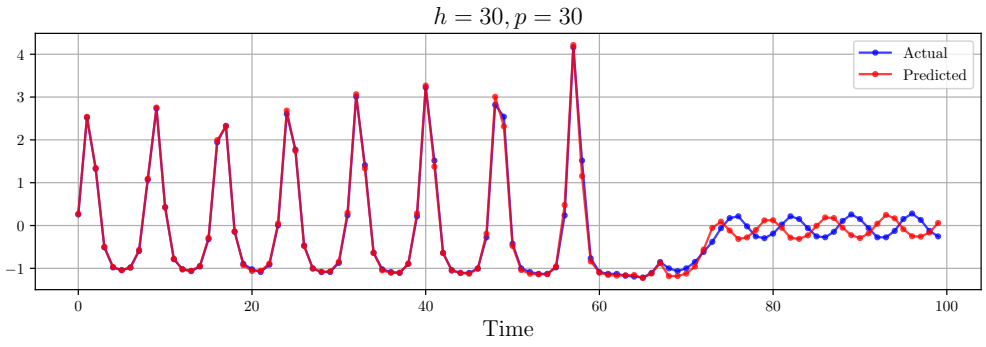

Figure 2.5: Predictions on the test set of an LSTM with 30 hidden units and a lag value of 30.

3 Deep feature learning

In this assignment, we will delve deeper into the concepts of deep feature learning. More specifically, we will investigate principal component analysis (PCA), stacked autoencoders and convolutional neural networks (CNN).

3.1 Principal component analysis

We discuss a few applications of PCA by considering the explained variance (the fraction of the cumulative sum of sorted eigenvalues over the total sum) and the root-mean-square deviation (RMSD), defined as $\sqrt{\text{MSE}}$. Figure 3.1 shows t-SNE plots of the three datasets of interest, from which we can get more intuition regarding the structure or randomness of the dataset.

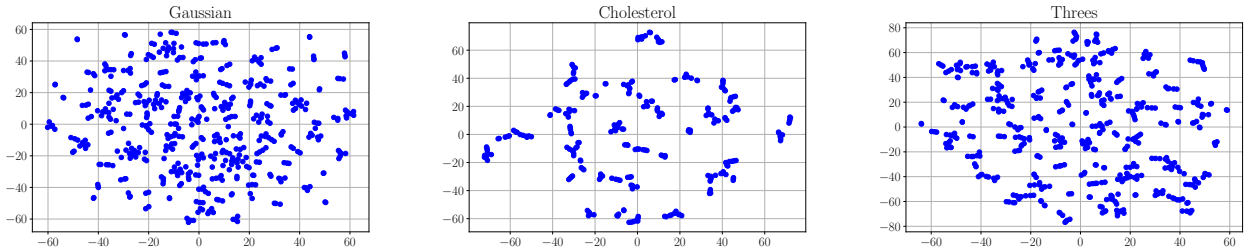

Figure 3.1: t-SNE plots of the three datasets.

First, we show the PCA analysis on 500 samples of 50-dimensional random Gaussian numbers in Figure 3.2. Due to significant randomness in the data, the eigenvalues are roughly equal in magnitude and we need around 40 out of 50 components to explain 90% of the variance of the dataset, which is quite high. Second, the PCA analysis of the cholesterol dataset is shown in

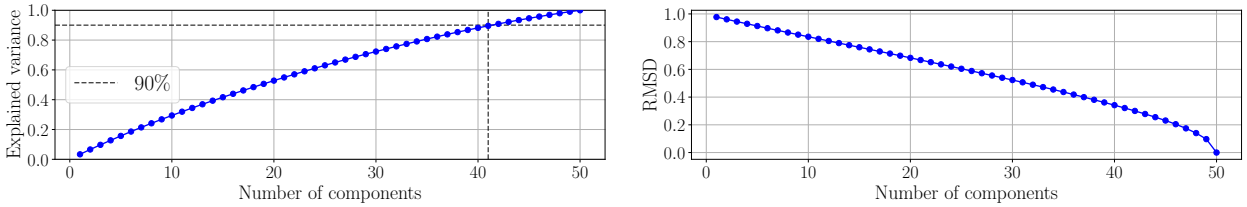

Figure 3.2: Analysis of PCA performed on random data drawn from a Gaussian distribution.

Figure 3.3. Since the data is highly correlated, we have only a few large eigenvalues and we can explain around 99.9% of the variance in the 21-dimensional dataset using only 3 components. As a final example, we analyze PCA on a dataset containing handwritten threes. The mean of the dataset is shown in Figure 3.5. Curiously, the reconstruction error for $q = 256$ components is not precisely zero, but around 10^{-5}. This is likely due to rounding errors, since many of the principal components are very close to zero. We consider the reconstructions of a few threes with varying number of principal components q, shown in Figure 3.5. For $q = 1$, the reconstructions of the different examples all look very similar, which clearly is a poor reconstruction. If we increase the

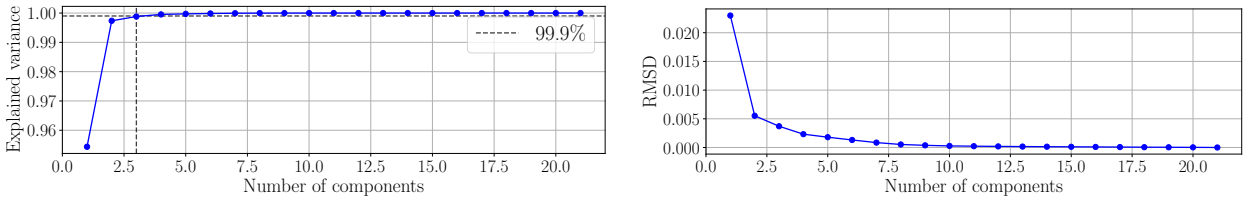

Figure 3.3: Analysis of PCA performed on highly correlated cholesterol data.

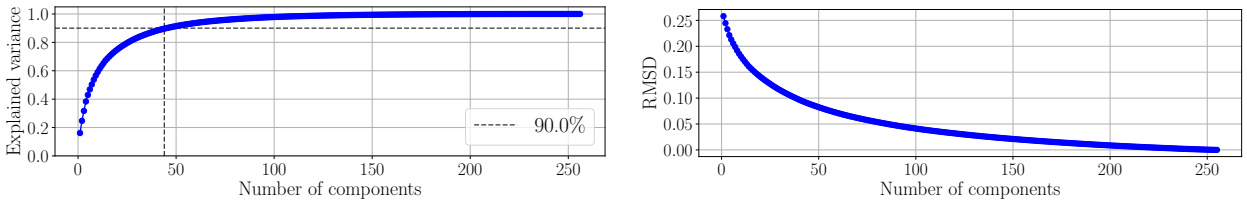

Figure 3.4: Analysis of PCA performed on handwritten digits.

number of principal components, more information is taken into account in the compression. As a result, the specific characteristics of each of the examples start to appear if we increase the number of components. As closing remark, we note that the above applications all demonstrate

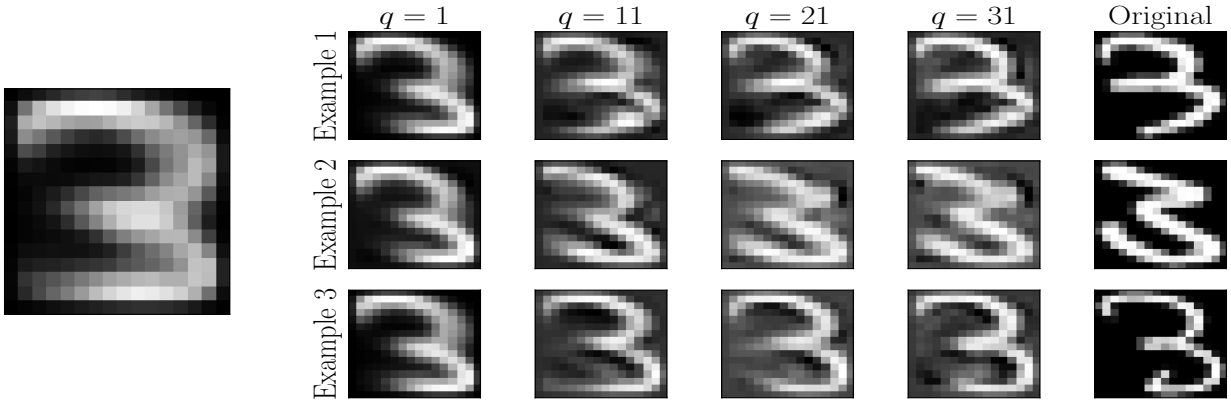

Figure 3.5: Left: Mean three. *Right*: Reconstructions of digits using PCA.

that the reconstruction error induced by ignoring a certain principal component is proportional to the magnitude of the corresponding eigenvalue.

3.2 Stacked autoencoders

We investigate classifiers for handwritten digits based on stacked autoencoders (SAE) with a softmax layer at the end. We restrict ourselves to varying the number of hidden layers and their sizes and train each configuration 3 times to smoothen out stochastic effects. The accuracy of each architecture is reported in Table 3.1. First of all, we notice that all architectures achieve an accuracy of around 99%, such that the classification seems quite easy to solve, especially given that each

class occurs with 10% probability such that a basic classifier predicting the majority class gives a baseline accuracy of only 10%. We find architectures that improve upon the standard set-up by increasing the number of layers, although a single hidden layer with 200 neurons also offers improvement. However, we noticed that the accuracy before fine-tuning is consistently worse if the architecture has more layers, with the only exception being the architecture with four hidden layers. Hence, fine-tuning gives significant improvements for these architectures. This is to be expected, since we train each layer separately in a greedy manner. Hence, stacking these layers after the greedy training is not guaranteed to give an optimal performance, especially if many layers are stacked. However, the autoencoders provide a good initial condition to start from, and fine-tuning guarantees that the stacked architecture performs well in the end despite the greedy training.

We compare these results against a feedforward MLP with two hidden layers of size 100 and 50 respectively and with a softmax output layer. Due to the large input dimensionality, we can not use the `trainlm` algorithm, as it requires too much memory for storing the Hessian matrix. Hence, we train with `traincgf` as this method has the best performance out of all algorithms that only rely on gradient information (see Figure 1.1). Training for 200 epochs with early stopping on 20% of the training data as validation set, we find an accuracy of 92.09%, averaged over 3 runs. Hence, training and stacking the autoencoders proved to be an improved initialization scheme compared to random initialization of the weights. Moreover, the results obtained that way are more robust, since the standard deviations of the values reported in Table 3.1 were all below 1%, while the accuracy of the MLP had a standard deviation of 6.80%.

Table 3.1: Accuracy of stacked autoencoders trained on handwritten digits, averaged over 3 repetitions and sorted based on accuracy after fine-tuning. The default set-up is highlighted in gray.

Hidden layers	Accuracy (greedy)	**Accuracy (fine-tuning)**	Improvement
$(200, 100, 50, 25)$	99.49%	99.65%	0.16%
(200)	99.29%	99.45%	0.16%
$(200, 100, 50)$	89.05%	99.35%	10.30%
$(100, 50, 25)$	67.59%	99.27%	31.67%
$(100, 50)$	90.77%	99.15%	8.38%
(100)	98.49%	99.01%	0.51%
$(200, 100)$	98.38%	98.71%	0.33%

3.3 Convolutional neural networks

For this exercise, we will investigate the AlexNet architecture in Matlab. We will inspect the first five layers of the network. The first layer is an input layer taking color images of shape $(227, 227)$. The subsequent layers learn appropriate responses to images. The second layer, which is the first convolutional layer, will mainly respond to basic features, such as shapes, blobs, edges, contrasts, gradient differences *et cetera*. This can be seen by plotting the filters, since they are the weights of this layer and hence determine the amount of activation that a pixel at a specific location of the kernel introduces in the layer. Inspecting this layer, we notice that it consists of 96 kernels of size $F = (11, 11)$, with stride $S = (4, 4)$ and padding $P = (0, 0)$. The size of the output of the kernel

can be computed with the formula [4]

$$\left(w^{(i+1)}, h^{(i+1)}\right) = \left(\frac{w^{(i)} - F_w + 2P_w}{S_w} + 1, \frac{h^{(i)} - F_h + 2P_h}{S_h} + 1, \right) , \tag{3.1}$$

where w, h stand for width and height and the superscript is a layer index. Hence the images have size $(55, 55)$ and there are 96 channels in the second layer. The third layer is a ReLU layer and the fourth layer is a cross channel normalization layer. The fifth layer is a max pooling layer, with kernel size $F = (3, 3)$, stride $S = (2, 2)$ and padding $P = (0, 0)$. Using Equation (3.1), we find that the output of the fifth layer, and hence the input of the sixth layer, has size $(27, 27)$ with 96 channels. The first fully connected layer is layer 17. The input comes from 256 channels of 6×6 arrays, which get flattened to a single array of size 9 216 for the fully connected layers. The original input dimensionality was $3 \cdot 227^2 = 154\ 587$. Hence, the input size that is fed into the classifier is less than 6% of the original input dimensionality. Clearly, this is a drastic improvement compared to the alternative where the original images, flattened to a 1D array, are used as input for the classifier.

Since CNNs were specifically designed to improve image classification with deep learning, they have a few advantages compared to fully connected networks. First, CNNs can learn the appropriate features from images by using convolutional layers. This leads to an improved accuracy in image classification compared to fully connected networks that treat each input pixel as a separate feature or are trained on features extracted using *e.g.* PCA or autoencoders, which can reduce dimensionality but may not learn the optimal features to use. This allows the CNN to train the classification part of the network on a rich, lower-dimensional representation of the images. Second, CNNs use pooling layers to further reduce the dimensionality of features and make the network more tolerant to noise to improve the robustness of the network. Third, the convolutional layers allow the network to generalize over variations in the position or rotation of features within the image. Fourth, CNNs are computationally efficient, since they reuse learned filters across the entire image and, in doing so, share the weights instead of learning separate parameters for each input pixel, making them faster to train and evaluate.

We investigate some CNN architectures trained on handwritten digits. A first observation is that these architectures are much harder to train compared to the MLPs studied before in this report, since even the basic provided architecture already has 7 hidden layers with over 22 000 trainable parameters and takes around 9 minutes to train 15 epochs. This architecture achieves an accuracy of 82.32%. Increasing the amount of convolutional layers did not seem to give significant improvements. This is likely due to the fact that the images are already quite simple and have a small size, such that more convolutions do not result in additional, useful information. Introducing another fully connected layer similarly decreased the accuracy to 79.69%. However, we can improve the performance by increasing the number of filters. For instance, by doubling the number of filters of both convolutional layers (to 24 and 48, respectively), we were able to achieve 100% accuracy already after 750 iterations. We also noticed that some architectures can lead to an oscillating behaviour in the accuracy at higher number of iterations, such that it would be advisable to either change the size of the mini-batches or employ an adaptable learning rate through a scheduler.

4 Generative models

In this assignment, we will dig deeper into generative models such as (deep) restricted Boltzmann machines, generative adversarial networks and optimal transport.

4.1 Restricted Boltzmann machines

The first architecture we investigate are restricted Boltzmann machines (RBM). We train an RBM on the MNIST digits dataset and visually evaluate the performance of the networks for reconstructing images. The number of epochs and the number of components determine the performance similar to standard MLP architectures. Increasing the number of hidden components allows the network to learn more complex latent representations of the digits, but can lead to overfitting. Increasing the number of epochs can improve the performance, provided that the learning rate is tuned adequately. We noticed that a low number of hidden components leads to noisy, sometimes even unrecognizable versions of the digits. Therefore, it is recommended to keep this hyperparameter high, such that we stick to the default value for the discussion. When sampling from the RBM, increasing the number of Gibbs steps can in some cases cause the RBM to generate different digits after sufficient iterations, as seen from Figure 4.1. This is likely since the randomness introduced by the sampling procedure can accidentally create features which cause the Gibbs sampling to converge to another digit. For instance, the bottom part of the 5 in Figure 4.1 is not very pronounced, such that after a few sampling steps, it almost completely disappears and the remaining features cause the network to recognize it as a 6. Other digits, such as 1, seem to be robust against this problem since they have very clear and simple features. When removing parts of the images, the quality of

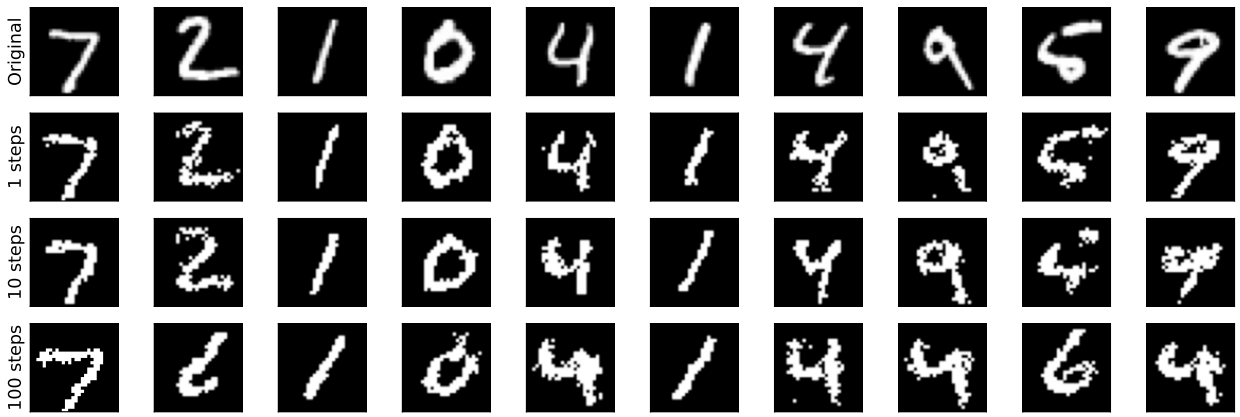

Figure 4.1: Gibbs sampling from the RBM

the reconstruction similarly depends on the hyperparameters as described above. The quality also depends on how much information we remove, or at which location. For instance, removing the top part of a 7 leaves only a pronounced diagonal line. Such a distortion creates an image which drives the RBM to converge to a 1 instead. On the other hand, removing its bottom part will likely leave sufficient characteristics such that the network can correctly reconstruct the digit into a 7 again.

4.2 Deep Boltzmann machines

Compared to RBMs, which have only a single layer of hidden neurons, deep Boltzmann machines (DBM) have multiple hidden layers. Similar to regular MLPs, this allows the network to learn a hierarchy of diverse hidden representations at different levels of abstraction. This is understood by comparing the components of the RBM of the previous section with those of the first two hidden layers of the DBM architecture, shown in Figure 4.2. Besides spots of high or low intensity, the RBM components also respond to curves and lines, and some components even vaguely resemble digits. Overall, these components are highly complicated as they are the only layer of latent features in the network. The components of the DBM, on the other hand, are much more basic but therefore also more abstract to interpret. The first layer seems to pick up information from spots, lines and gradients, while the second layer seems to have learned information regarding contrast differences, especially close to the boundaries of the images. The fact that DBMs have multiple

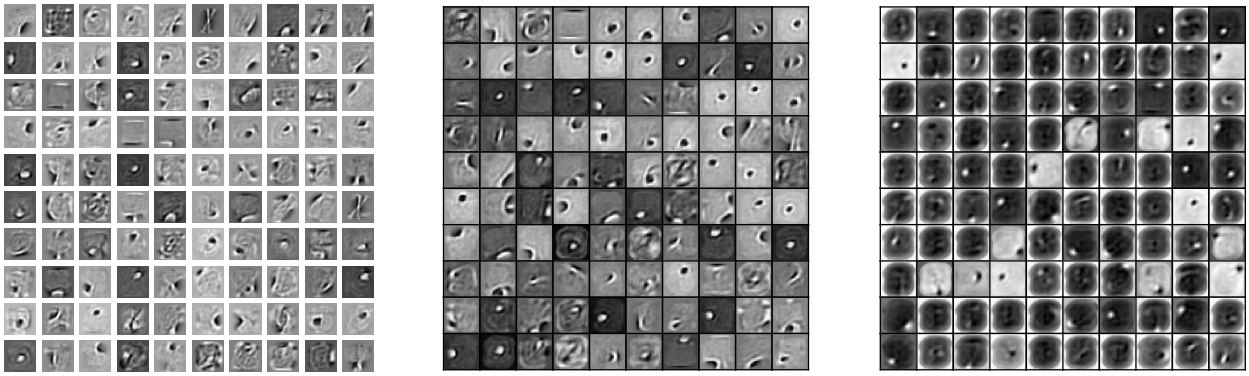

Figure 4.2: Left: Components of the RBM. *Middle*: Components of the first layer of the DBM. *Right*: Components of the second layer of the DBM.

layers leads to an improved performance, as can be seen from the samples in Figure 4.3. Moreover, the generated digits are much more coherent compared to those of RBMs and we have less noisy, random pixels. This is because an architecture with several hidden layers can not only learn which features are important, but also learns which combinations of features are important. This is clearly an advantage in the context of (re)constructing digits.

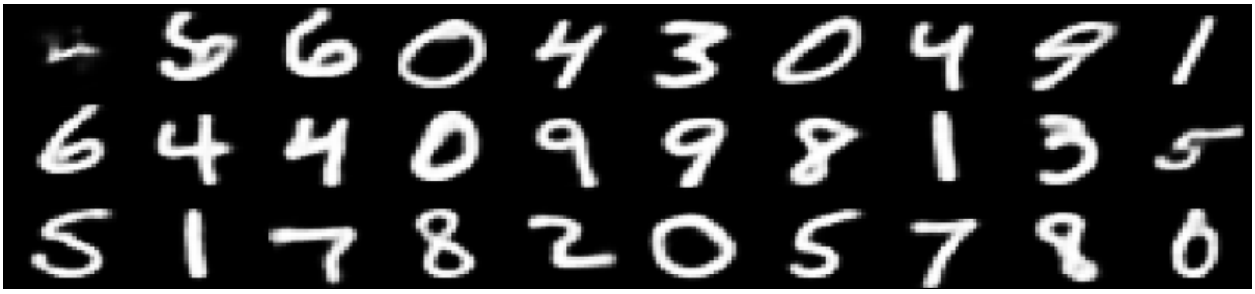

Figure 4.3: Samples from the DBM.

4.3 Generative adversarial networks

The next generative model we study are generative adversarial networks (GANs). We trained a deep convolutional GAN (DCGAN) on pictures of dogs from the CIFAR dataset. The loss and accuracy during the final 500 training epochs is shown in Figure 4.4. Since the discriminator's accuracy oscillates around 0.5, the DCGAN has likely converged. However, the losses of the networks are still oscillating wildly around their mean values and it is clear that this equilibrium is unstable and hard to maintain during training. At some points, such as around 250 epochs, the networks can be significantly disturbed from their equilibrium. This unstable balance is due to the fact that the discriminator's feedback becomes less informative over time, eventually becoming almost random [5]. We also note that the figure clearly shows that the training of GANs is based on a minimax approach. That is, the discriminator and generator have their peaks on opposite sides of their mean value, such that the generator overperforms when the discriminator underperforms and vice versa.

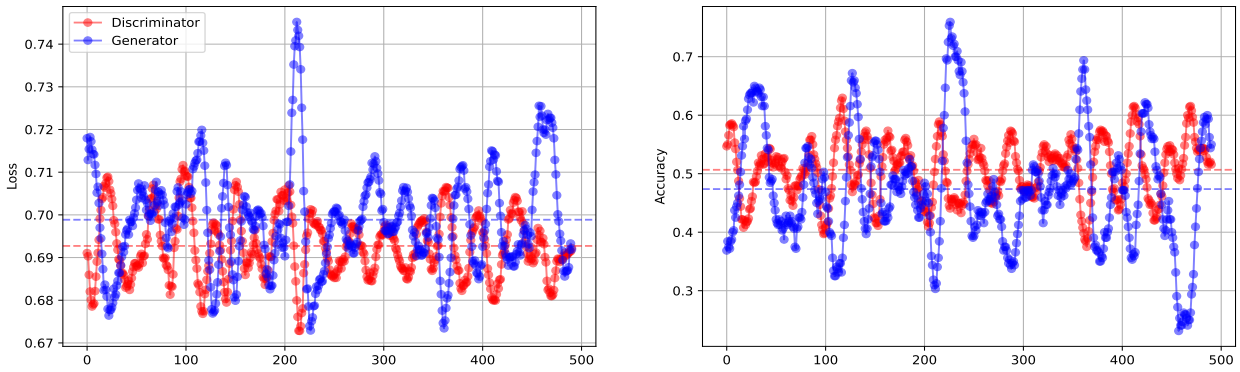

Figure 4.4: Final values for the loss (left) and accuracy (right) of the discriminator (red) and generator (blue) of training the DCGAN. Dashed lines show average values over the plotted interval.

4.4 Optimal transport & Wasserstein GANs

Finally, we investigate the theory and applications of optimal transport (OT), which defines a metric between probability distributions, the Wasserstein metric. As a first application, we explore the optimal transport between the two images in Figure 4.5 which display a very similar scenery but with different colours. We also plot a kernel density estimate plot of the intensity values of the red, green and blue pixels of both images to visualize the probability distributions of interest. The results of the optimal transport are shown in Figure 4.5, using the Wasserstein and Sinkhorn distance. The Sinkhorn distance adds a regularization term on the transportation plan to the Wasserstein distance, resulting in images with smoother brightness differences. The crucial aspect of this transportation is that groups of pixel values retain their meaning. That is, while the transported image of "ocean day" has the same color distribution as the original "ocean sunset" image and hence looks much redder than its original, we can still recognize distinctive features from the original picture, such as the single cloud in the middle. Hence, while OT modified the

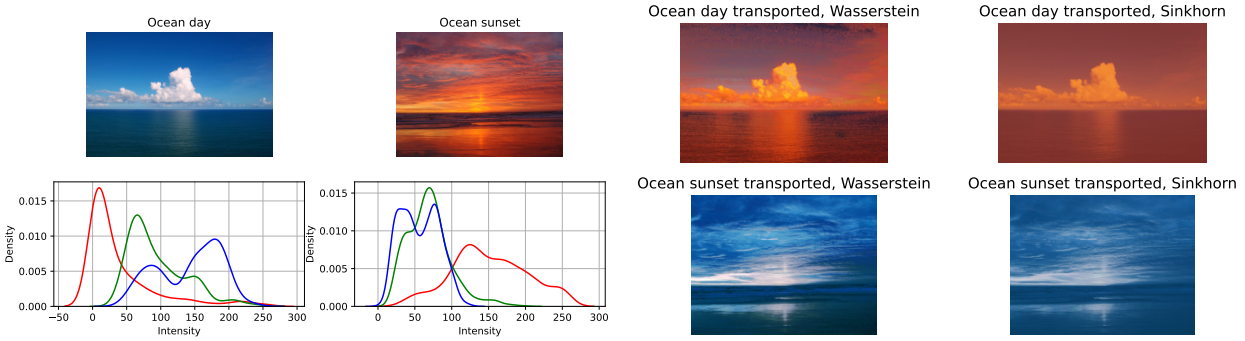

Figure 4.5: Left: Original images and their color distributions. *Right*: Transported images, using the Wasserstein and Sinkhorn distance.

color distributions, it respects spatial correlations among pixel values such that we are still able to recognize the original images in the transported ones. Moreover, the transition between images that OT provides is a smooth transition, as explained in the assignment. Clearly, these key properties are not shared by an alternative which simply swaps all individual pixels.

Wasserstein GANs (WGANs) are GANs that use the Wasserstein distance to measure distances between probability distributions. Using the Wasserstein distance instead of the Kullback-Leibler divergence, which standard GANs use, has some advantages for training [6]. For example, it has been shown to be a better measure of distance than *e.g.* the KL divergence because it is continuous and differentiable, such that the gradient is more stable. Also empirically, it has been observed to give a more robust way to train GANs. We show this by plotting the final training epochs of a WGAN trained on the MNIST digits dataset in Figure 4.4, which should be compared to the training of a DCGAN, for example in Figure 4.4. The training is more stable, especially for the discriminator, since this network is updated more often than the generator at each iteration of the training [6, 7]. The target distribution is more accurately captured than in the standard GAN context such that the generated samples (not shown here due to space limitations) have a much higher quality. Moreover, as a result of the different training objective, WGANs are less prone to suffer from mode collapse [8].

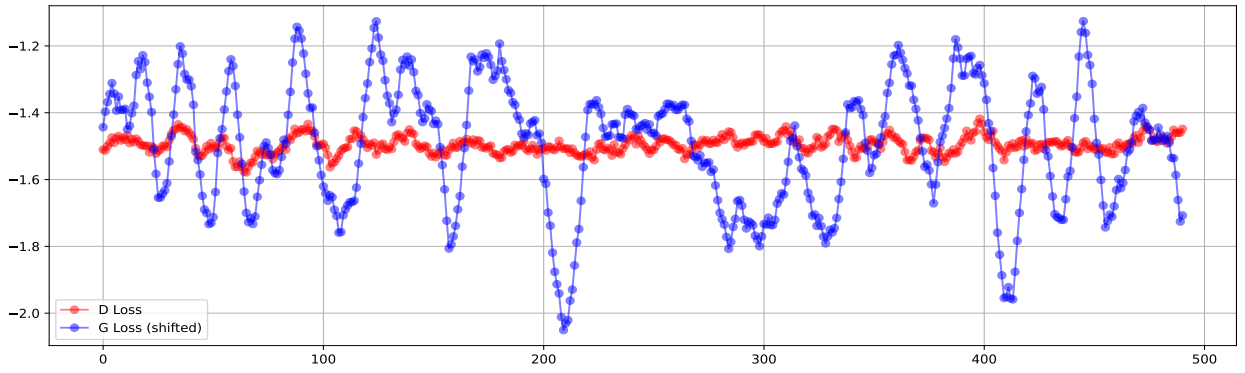

Figure 4.6: Final values for the loss of the discriminator (red) and generator (blue) of training the WGAN. The loss values of the generator have been vertically shifted to enhance comparison between the plots.

References

[1] Mathworks. *Bayesian regularization backpropagation - Matlab* ***trainbr***. https://nl.mathworks.com/help/deeplearning/ref/trainbr.html. Accessed: 2023-05-08.

[2] Jason Brownlee. *Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras*. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/. Accessed: 2023-05-08. 2016.

[3] Mathworks. *Time Series Forecasting Using Deep Network Designer*. https://nl.mathworks.com/help/deeplearning/ug/time-series-forecasting-using-deep-network-designer.html. Accessed: 2023-05-08.

[4] Android Knowledge Transfer. *Calculate Output Size of Convolutional and Pooling layers in CNN*. https://androidkt.com/calculate-output-size-convolutional-pooling-layers-cnn/. Accessed: 2023-04-19.

[5] Thomas Wood. *Generative Adversarial Network Definition*. https://deepai.org/machine-learning-glossary-and-terms/generative-adversarial-network. Accessed: 2023-05-27.

[6] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks". In: *International conference on machine learning*. PMLR. 2017, pp. 214–223.

[7] Jason Brownlee. *How to Develop a Wasserstein Generative Adversarial Network (WGAN) From Scratch*. https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wgan-from-scratch/. Accessed: 2023-05-02.

[8] Haozhe Liu et al. *Combating Mode Collapse in GANs via Manifold Entropy Estimation*. 2023. arXiv: 2208.12055 [cs.CV].