FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
MASTER OF ARTIFICIAL INTELLIGENCE

KU LEUVEN

Cumulative Report on Artificial Neural Networks

Based on the Exercise Sessions 1,2,3,4

Antonios Glioumpas (r0827913)

Spring Semester 2022

Contents

Part I

Supervised learning and generalization

1 Backpropagation in feedforward multi-layer networks

1.1 Function approximation: comparison of various algorithms

Given the function $y = sin(x^2)$ we want to compare how different neural networks trained with different learning algorithms perform in approximating it. The goal here is to compare the performance of a network when training it using gradient descent (traingd) as training algorithm, compared to various other training algorithms:

The workflow followed to compare these training algorithms is described as follows: Eight feedforward networks with 50 hidden layers are created. All input data are used to train the network (no validation and test split takes place at this stage). In each run (training session for 500 epochs), the weights and biases of all networks are randomly initiated with the same weights and biases.

The comparison of the results is made by analysing: the **number of epochs/best epoch** until convergence, the **Mean Squared Error (MSE)** related to the difference between the learned function and the ground truth value. The **r-value (R) - correlation coefficient** associated to the learned function. Each run consisted of 500 epochs, and was repeated 30 times, to reduce the variability due to the random initialization. Table 1 illustrates the average values for these 30 runs, each with 500 epochs, for the algorithms under comparison.

Training algorithm	Best epoch	MSE	r (correlation coefficient)	Time (sec)
1) traingd	500	0.418	0.433	0.715
2) traingda	477	0.178	0.792	0.647
3) traingdm	500	0.416	0.436	0.673
4) traingdx	497	0.065	0.931	0.653
5) traincgf	302	0.001	0.998	1.131
6) traincgp	454	0.001	0.998	1.180
7) trainbfg	500	0.00043	0.999	3.313
8) trainlm	500	0.00013	1	2.184
9) trainbr	500	0.00004	1	2.101

Table 1: Mean values showing the performance of 8 different training algorithms on 189 noiseless datapoints for 500 epochs, 30 runs. **traingda** : Gradient Descent with Adaptive Learning Rate **traingdm** : Gradient Descent with momentum **traingdx** : Gradient Descent with momentum and adaptive learning rate **traincgf** : Fletcher-Reeves conjugate gradient algorithm **traincgp** : Polak-Ribiere conjugate gradient algorithm **trainbfg** : BFGS quasi Newton algorithm (quasiNewton) **trainlm** : Levenberg-Marquardt algorithm (adaptive mixture of Newton & Steepest Descent alg.) **trainbr** : Bayesian Regularization backpropagation algorithm

Observations:
The conjugate methods (Quasi-Newton and Levenberg-Marquardt) give better results, however they need more time to be executed. The average MSE in their best epochs is also lower. When stopping the training when a certain MSE is obtained, methods such as Levenberg-Marquardt finish sooner, since they perform better than gradient descent methods. Moreover, the approximation of the function with the networks trained with the conjugate methods gives a better correlation coefficient.

Features, pros and cons of each learning algorithm :
The way to know if the networks are learning properly is by measuring the difference of the model's output with the ground-truth. The Mean Squared Error (MSE) allows us to calculate that difference. Then, we can define it as a cost function to minimize during training. Based on that cost, the algorithms using gradient calculations give the direction for updating the interconnection weights of the network:

- **Gradient descent (traingd)**: Used to update the parameters of the model (weights and biases) by optimizing the function that indicates the steepest descent direction (negative of the gradient). Gradient descent keeps the learning rate (step size) constant, which makes the network very sensitive to this hyperparameter (low value: longer to converge; high value: oscillations and instability). In Table 1 it is evident that has the poorest performance of all algorithms. (needs the maximum number of epochs to reach its best epoch, has the highest MSE and the worst correlation coefficient).
- **Gradient descent with adaptive learning rate (traingda) and with momentum (traingdm)** : **adaptive learning rate**: When the learning rate increases when the error in a training epoch decreases (and vice versa), this leads to an improvement of the overall performance of the Gradient Descent algorithm as less epochs are needed until the best epoch is reached, the MSE is lower and the correlation coefficient higher. **Momentum**: The modified Gradient descent with the addition of a momentum term leads to similar result as with the simple Gradient descent, the experiment showed no significant improvement in the task at hand. **Adaptive Learning Rate & Momentum**: Leads to the best results concerning the Gradient Descent Method.

Other ways to optimize the minimization of the energy/cost function include the **Newton method**, an iterative algorithm that finds the best search direction based on the Taylor expansion. However, it requires calculating the inverse of the Hessian matrix (the second order derivatives), which is computationally expensive. Based on Newton's method, other algorithms that avoid the Hessian implementation have been proposed:

- **Quasi Newton (trainbfg)**: Is an approximation of the Hessian matrix, that imitates its properties considering only the first order derivatives using rank 2 updates. The experiments showed that comparatively with the other seven methods, this one had the second best results.
- **Levenberg-Marquardt (trainlm)**: For ill-conditioned matrices (especially in the case when there are many unknown parameters), it is more difficult to calculate the inverse of the Hessian which results in a performance decrease of the Newton's method. The use of a Lagrangian in this method allows us to have an interpolation between Newton and steepest descent methods. It adds a constraint in the step size that optimizes, where having a small λ parameter, will be similar to Newton and converges faster; with a larger value, it will be slower and similar to steepest descent. The experiments showed that this method, comparatively with the other seven, had the best performance.

Conjugate gradient methods address the problem of the huge storage that Newton method would require for storing the Hessian by searching for the line that will give the optimal distance: it starts considering the steepest descent direction, and then the conjugates of the previous directions. Two known of these methods are: **Polak-Ribiere (traincgp)**, which uses four vectors to store, and **Fletcher-Reeves (traincgf)**, which uses three vectors.

1.2 Learning from noisy data: generalization

Observations: The same pattern as in the noiseless data scenario is observed here. However, the performance of all the algorithms is worse, due to the added noise, which makes identifying the underlying function (generator) of the data more difficult to approximate. It should be noted that after 500 epochs gradient descent performs better in the case of noisy data. By increasing the number of available data, an increase in performance is observed, with the exception of GD, which in this case requires more than 500 epochs to reach maximum accuracy (for a specific training algorithm, for more data points, more epochs are required to increase accuracy).

1.3 Personal Regression example

After multiple trial and error efforts, the following network was chosen and trained for a max 500 epochs:

Training Algorithm	189 datapoints, $\sigma = 2\%$		472 datapoints, $\sigma = 2\%$		472 datapoints, $\sigma = 4\%$	
	MSE	**r**	**MSE**	**r**	**MSE**	**r**
1) traingd	0.412	0.512	0.412	0.502	0.463	0.624
2) traingda	0.253	0.724	0.243	0.732	0.421	0.672
3) traingdm	0.409	0.516	0.409	0.506	0.459	0.629
4) traingdx	0.104	0.909	0.096	0.926	0.207	0.912
5) traincgf	0.017	0.978	0.032	0.992	0.128	0.969
6) traincgp	0.017	0.978	0.032	0.992	0.129	0.970
7) trainbfg	0.014	0.976	0.031	0.992	0.123	0.962
8) trainlm	0.009	0.971	0.028	0.990	0.114	0.955
9) trainbr	0.0178	0.982	0.029	0.991	0.1298	0.964

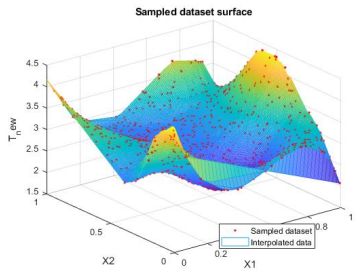
Table 2: Performance of different learning algorithms with various numbers of datapoints and Gaussian noise distributions. 30 runs, 500 epochs of training in each. Mean values for MSE and r (correlation coefficient), σ : Standard deviation of added Gaussian noise, r : correlation coefficient

Number of hidden layers = 2 Given the non-linearity of the target vs the feature space, 2 hidden layers instead of 1 were selected. It was also found that many more neurons are required in a 1-hidden-layer architecture vs a greater-than-one hidden layer architecture in order to reach similar accuracy. Since more neurons would be required, this translates to more weights and biases to optimize, hence increased number of epochs.

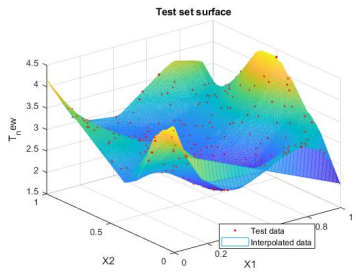
Number of neurons = [10 ; 5] 10 neurons for the 1st hidden layer and 5 for the 2nd, since the neuron number decreases as hidden layers progress. The 2:1 ratio was arbitrarily chosen. It was found that using a lesser number of neurons would decrease the accuracy for a specified number of epochs.

Train algorithm = 'trainlm'. The advantages of the LM algorithm decrease as the number of network parameters increases. As the number of parameters increases, the computation required in the LM algorithm increases geometrically.

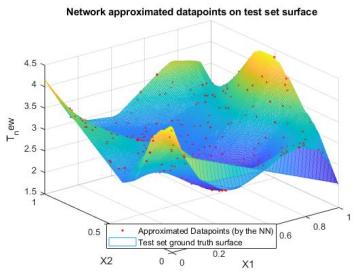
Transfer function = 'logsig' for the hidden layers' neurons, 'purelin' for the output layer. Log-sigmoid transfer function was used for the hidden layers, instead of the default tan-sigmoid, because it was found to lead to lesser number of validation checks. Purelin transfer function was used for the output layer, as is the case for function approximation problems.



(a) 1000 Sampled data points plotted on top of the surface they define



(b) Test set datapoints plotted on top of the surface they define



(c) Network approximated outputs plotted on top of the surface defined by the test set data

Apparently, the model is a very good fit for the data. Training, validation and test sets have very similar errors, and no overfitting occurs (i.e. training error does not continue to decrease whereas test error increases). Training error is only slightly lower compared to the test set error, which is expected, since that is the error which the network is being optimized to minimize. However, all errors are in the same order of magnitude.

Ways to improve performance: In terms of **accuracy**, the results are already very good. However, if we let the network run for more epochs, error can be further minimized. In terms of **simulation time**, already model is pretty fast. However, other network architectures can be tested (more hidden layers with less total neurons in the network) to check whether solution converges faster, thus requiring less epochs to reach the same error.

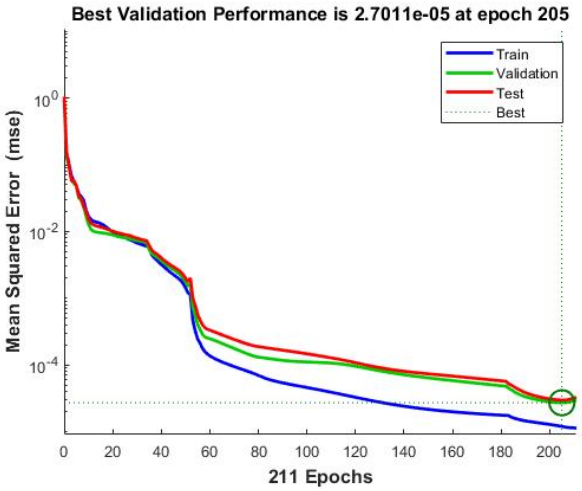


Figure 2: Train, Validation, Test set error curves based on the MSE. Mean Squared error of the test set = 0.0000295 at epoch 205.

2 Bayesian Inference of Network hyperparameters

Instead of using early stopping, one way to apply regularization is by Bayesian learning. The results are shown at the end of Table 1, and are quite similar to Levenberg-Marquardt, except that execution time is longer. For noisy datapoints, the results are still better than steepest descent, but worse than the results yielded by using the rest of the training algorithms. However, with the presence of more data, performance improves, especially when we take R into account.

One advantage of using this training method is that there is no risk of overfitting, and thus, no need of validation set; this is because the method automatically selects the best hyperparameters (a,b).

With many neurons, an over-parameterized network does not show any improvement or deterioration, since the algorithm finds an effective number of parameters (less than the total number of parameters k). The error level curve in this case showed more instability. In general, it is better to run Bayesian inference a couple of times, since it can have multiple local minima solutions, due to the different initializations.

Part II

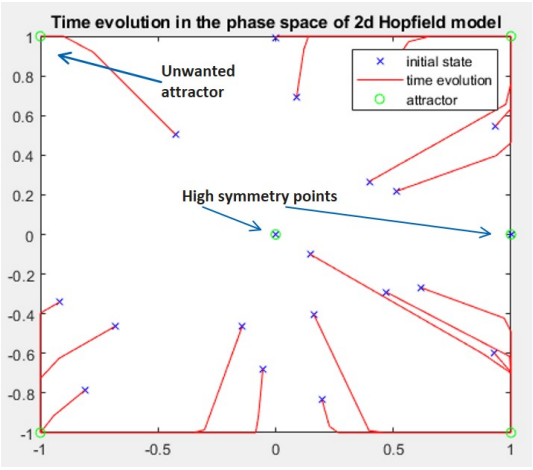
Recurrent Neural Networks

1 Hopfield Network

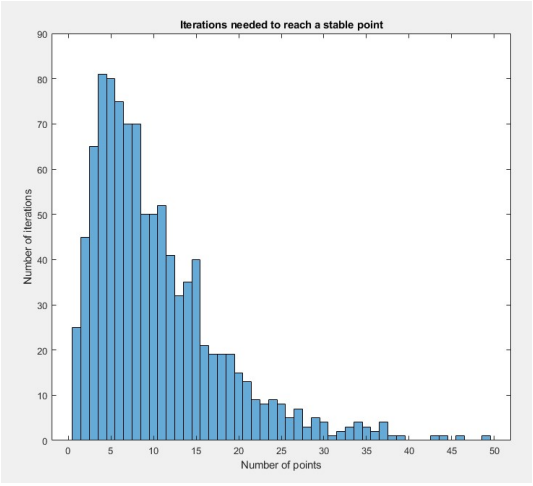
1.1 2-neuron Hopfield Network

The real attractors (the ones where the network stabilizes) are not the same as the ones used to create the network. We see that another attractor, symmetric with respect to the rest of the attractors, is created. That's because when storing an attractor, the symmetric one is stored as well. These unwanted stored patterns are called spurious states.

As shown in figure 3, it usually takes between 5-10 iterations for a point to reach an attractor. Points that are initially located in high symmetry positions (e.g. equidistant from all the attractors) will not be able to reach any attractor. Also, we observe that the points that are located closer to an unwanted attractor, will reach it instead of an actual one. The histogram in figure 3 illustrates the number of iterations needed to reach a stable state and the corresponding number of points that needed this number of iterations.



(a) Evolution in the phase space of a 2d Hopfield model. Total number of points: 20

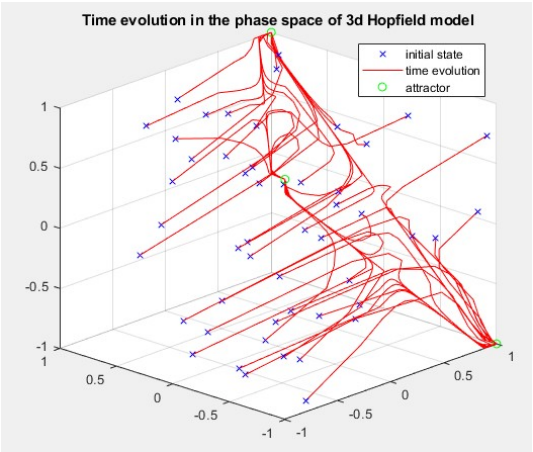


(b) Total number of points: 1000

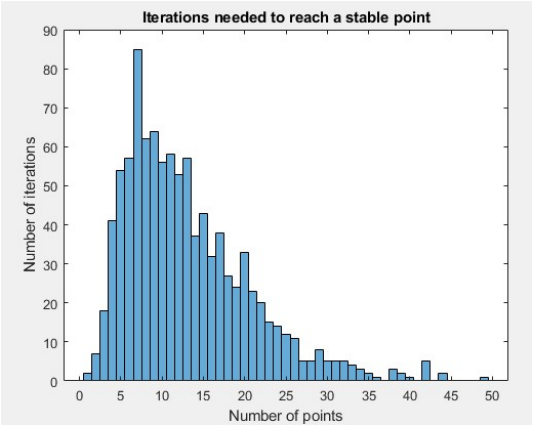
Figure 3: Using rep2.m to generate N random initial points and visualise simulation results of a 2d Hopfield network

1.2 3-neuron Hopfield Network

A 3D Hopfield Network behaves very similarly. By generating random points and letting them converge, we observe now that the network has been able to store correctly the 3 attractors that it was supposed to, without any additional attractor. This is due to the increased dimension of the network, which implies a higher probability of correctly storing a certain amount of attractors comparing to networks with less neurons. Intuitively, it seems unlikely to correctly store 3 attractors in a 2-neuron Hopfield Network. The average number of iterations required for convergence is slightly higher than before, due to increased dimensionality as observed in figure 4.



(a) Evolution in the phase space of a 3d Hopfield model. Total number of points: 50



(b) Total number of points: 1000, 3d Hopfield model

Figure 4: Using rep3.m to generate N random initial points and visualise simulation results of a 3d Hopfield network

1.3 Application to noisy images (function “hopdigit”)

In this part of the report, handwritten digits (0 to 9) were stored as equilibrium points of a Hopfield network. The ability of the network to retrieve these patterns from noisy data was evaluated.

The network is not always able to reconstruct the noisy digits, and that highly depends on the noise level. After a certain noise level, the distance of some noisy digits may come closer to a different attractor than the desired one, causing the noisy images to reach an unwanted attractor. In figure 7, for increasing noise levels, the number of iterations was successively increased until the noisy digits reach the original 'correct' attractor. If they failed to do so, another run was initiated and the number of iterations was increased. The results are shown up until noise level 9, because after that level, convergence became too difficult for the network (that is, to reach the desired attractors). It can be clearly seen that after noise level 7, it becomes very difficult to converge to the correct attractors.

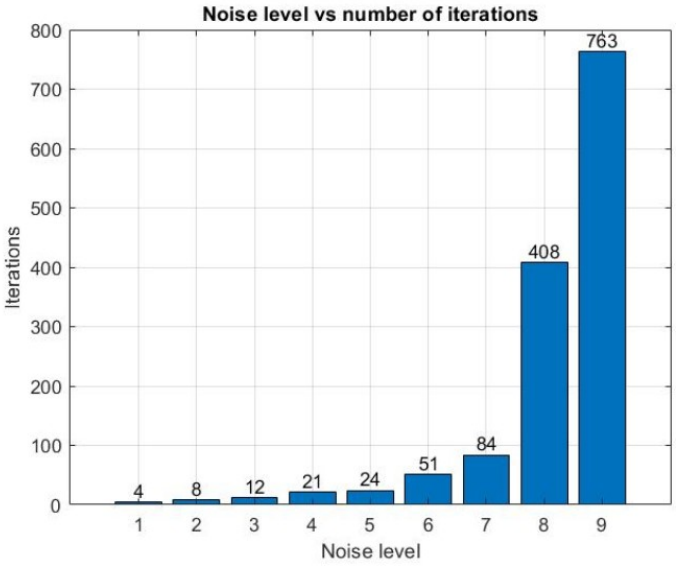


Figure 5: Number of iterations needed until the network converges to the original "correct" attractors

As a **conclusion**, higher noise levels greatly increase the chance of ending with a different attractor. In general, the number of required iterations is quite low, for example for a noise level of 3, only 10 iterations are enough to reach an attractor, on average.

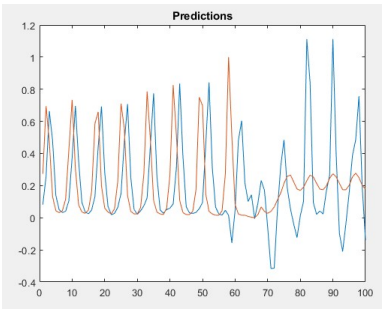
2 Time-Series Prediction

Various 1-hidden layer architectures of a Neural Network trained in a feedforward mode were investigated in order to predict the first 100 values of the time series by utilizing the last training set observations.

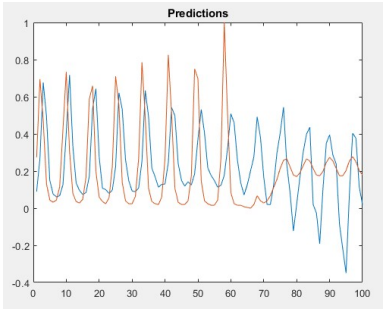
For the first prediction, the last p observations from the training set were used as inputs (p : model lag). For second prediction only the last p-1 observations from the training set plus the first prediction were used as inputs. Following the pattern above, one step at a time, 100 predictions were made. For the last prediction, only the very last training set observation was used. Then the ground truth values of the actual test set was used as comparison to calculate the test error.

Table 3: Most characteristic sets of values gained during the experiments

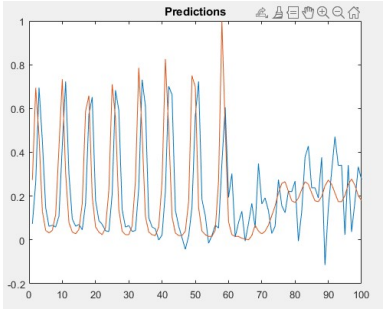
Models	lag	#neurons	MSE
1)	100	10	0.11968
2)	100	50	0.095552
3)	100	100	0.089857



(a) Model 1: **10 hidden layer neurons**, lag 100 observations



(b) Model 2: **50 hidden layer neurons**, lag 100 observations



(c) Model 3: **100 hidden layer neurons**, lag 100 observations

Figure 6: The ground truth values of the test set are illustrated with orange color and the predictions by the different models with blue color. All models were trained for **40 epochs**.

Details related to the model training:

- Only the "trainlm" (Levenberg–Marquardt) training algorithm was tested, as it is suitable (according to Matlab documentation) for function approximation of a few hundred weights.
- A lag of around 100 was assumed to achieve the best results, given the fact that the test set also contains 100 points. In the training set there is a section similar to the test set. Hence if part of the training contains the highest amount of input-output pairs very similar to the test (which implies a lag of 100), then this should lead to the best solution.

Having the MSE as a criterion for the suitability of the model to correctly predict the correct values and by visual inspection of the similarity of the blue with the orange curves in figure 6. The best results were obtained with an architecture of 100 neurons in the hidden layer and a lag of 100 observations.

3 Long Short-Term Memory Networks

The LSTM model is trained in a recurrent way, since it needs to know its current state in order to predict the next result, and in order to know its current state it has to have processed all the previous inputs.

At a high level, an LSTM cell works as follows:
Input gate: (together with the cell candidate): controls which input to add to the memory cell, that is inside the LSTM Network.
Forget gate: controls the amount of the previous state that is retained.
Output gate: controls the output of the network.

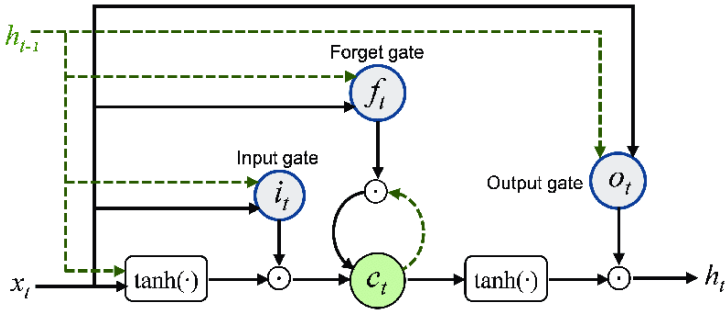


Figure 7: An LSTM cell structure showing the Input, Forget and Output gates.

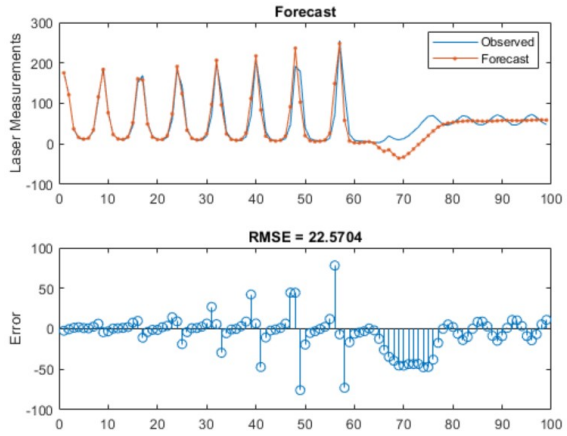
The output of each gate is then multiplied by the previous state (input gate) or by the state activation function of the current state (output gate). All these gates have as inputs both the current input and the previous output. Thus, this particular structure can work as a memory that learns what to retain and what to forget in order to correctly predict a time series. In our case, this architecture is suitable since there appear to exist "long" term correlations between values in the time series, but a high lag is unfeasible in terms of generalization and training set dimension (1000−lag).

The best results were gained by tuning the network with the following parameters: 500 training epochs, 100 neurons in the hidden layer, "Adam" training optimizer, Initial Learning Rate=0.005, Learning Rate Drop Period=125 epochs and Learning Rate Drop Factor=0.2.

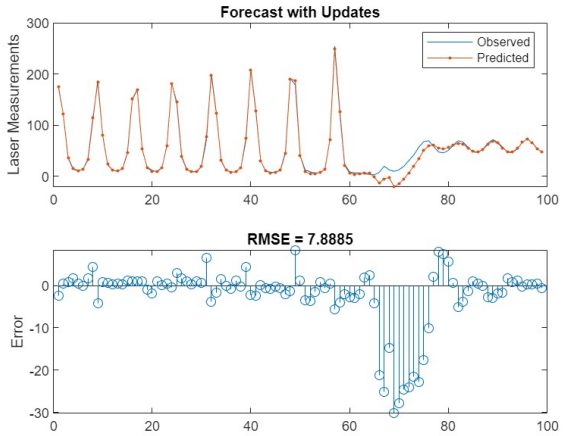
In order to create predictions based on the last observations of the training set, two cases were examined:

Case 1: The first prediction is made using the last time step of the training response. Here, we loop over the remaining predictions and input the previous prediction to gain the next prediction. It was observed that as the number of neurons in the hidden layer increased the predictions were closer to the ground truth values.

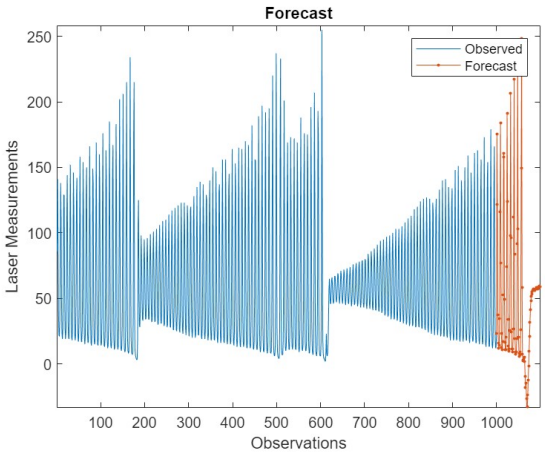
Case 2: We have access to the actual values of time steps between predictions. Therefore, we can update the network state with the observed values instead of the predicted values. As observed this strategy is less sensitive to the number of neurons used in the hidden layer.



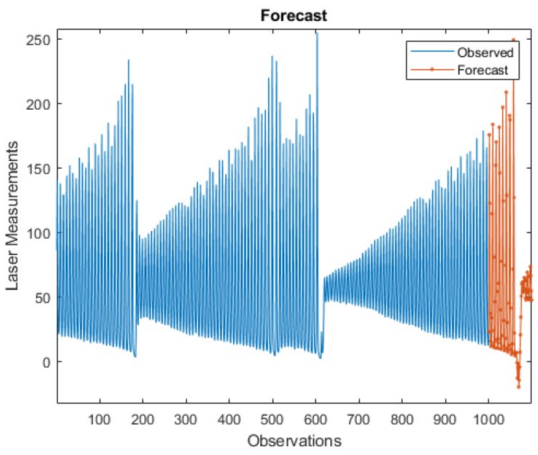
(a) Case 1



(b) Case 2



(a) Training set values & Forecast (Case 1)



(b) Training set values & Forecast (Case 2)

Conclusions: The RMSE for the prediction becomes lower as the lag increases, and the in the case of forecasting future timesteps without updating the network state, it is possible to predict the qualitative behaviour (value drop) correctly with sufficient lag. Comparing the results from the LSTM to the recurrent neural network, it is clear that the LSTM performs better. Furthermore, the LSTM takes less time to train. In order for the recurrent network to adequately capture the qualitative behaviour of the signal, it must be trained with a large number of inputs (large lag), whereas, in the case of the LSTM, we were able to accurately predict the signal using a lag value of one, and so the training time was much shorter. With a larger lag value, the RNN model achieves better performance. This is logical, since the model is making use of the relevant information, and with a wider window, it has more information at its disposal.

Part III

Deep Feature Learning

1 Principal Component Analysis

Mean vector of the dataset representing the "mean three": The mean three was computed using the command `imagesc(reshape(mean(threes),16,16),[0,1])`. The resulting image is shown in Figure 10.

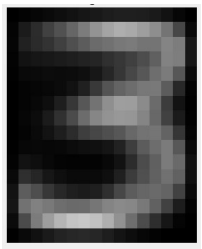


Figure 10: Mean "three"

Eigenvalues:
Next, the eigenvalues & eigenvectors are computed. The eigenvalues are seen in Figure 11. Based on the very small eigenvalues after 50 Principal Components (PC), it can be concluded that 50 components (restructured dimensions) should be sufficient to effectively reduce the original (265-dimensions) dataset.

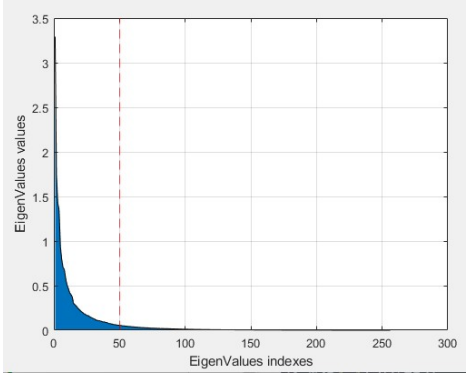


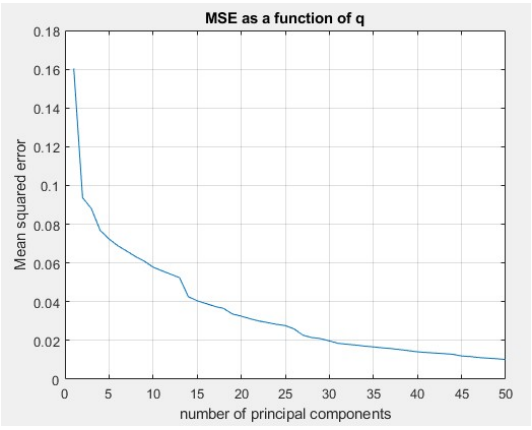
Figure 11: EigenValues plotted. The area below the curve is shaded for better visibility of the curve.

Reconstruction:
Then, based on the number of principal components, a part of the eigenvectors matrix was utilized as a projection matrix (e.g. for 1 PC, the resulting projection matrix would equal to the first column of the eigenvector matrix, for 2 PC, the projection matrix would equal to the first 2 columns of the eigenvector matrix etc...). The results are shown in Figure 12.

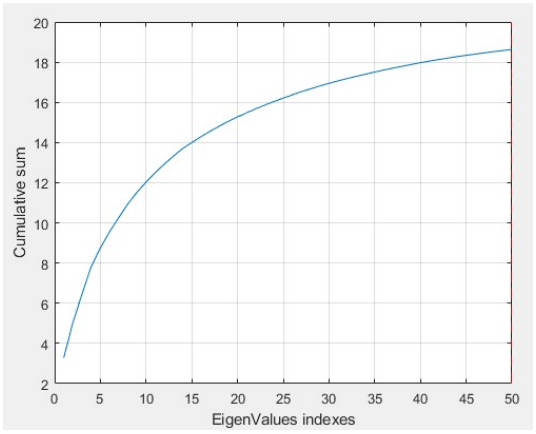
The assumption that the most important features are captured within the first 50 PC, based on the eigenvalues' curve, is validated. However, even with 2 PCs we observe that the majority of the features are captured, indicating the abstract shape of the number 3 symbol.

Figure 12: The two first "three" images of the dataset reconstructed for 1,2,3,50 principal components and the original image to the far right for comparison. Note that for 50 Principal Components, the reconstruction is very close to the original image.

Reconstruction Error:
In theory, the reconstruction error when using the full number of principal components, should be equal to zero. In practice, for q=256, the reconstruction error is MSE=5.657e-31. This small difference can be attributed to the propagated errors in the internal calculations during the PCA process, i.e., the mean, the covariance matrix calculation, the eigenvectors. All of these calculations imply operations that might induce slight loss of precision.



(a) Mean Squared Error between the original and the reconstructed dataset plotted as a function of q (number of principle components used for the reconstruction)



(b) Cumulative Eigenvalues Sum: Plot of the vector whose i-th element is the sum of all but the i largest eigenvalues for i = 1:1:50

Cumulative Eigenvalues:
Finally, the requested vector of cumulative sum of all but the 50 largest eigenvalues is plotted above. The shape of the curve validates the claim that the reconstruction error induced by not using a certain principal component is proportional to its eigenvalue. The lower the contribution of an eigenvalue to the cummulative sum, the lower the impact of its absence the the reconstruction error.

2 Digit classification with Stacked Autoencoders

Using stacked autoencoders we can define a deep network that encodes the input (reducing its dimensionality) and classifies it. Firstly, each autoencoder is trained in a feed-forward way, and after stacking their encoder parts and adding the softmax layer, using backpropagation at the stacked network, the model is fine-tuned. Then, a 1−hidden layer and a 2−hidden layer normal neural net are tested on the same task and the results are compared with the deep net.

The experiments, for the digits classification task, where different combinations of topologies and hyperparameters were tested can be found in Table 4.

Observations:

About **max training epochs**: A small increase in accuracy is observed in the pretrained deep network when increasing the number of epochs but not in the fine-tuned deep network.

Number of neurons (in the hidden layers): Reducing it leads to dramatic decrease in accuracy of the pretrained deep net, whereas increasing it has the opposite effect. Also, by doubling the encoders' size, we observe that the pretrained stacked model surpasses the normal networks' accuracies and reaches the accuracy of the fine tuned one. The fine-tuned deep net shows robustness against changes in hidden layer sizes.

Number of layers (added encoder layers): The original stacked network was compared with stacked networks of one added encoder(experiments in Table 4, lines 11) and 12)). The added encoders were trained with the same epochs as the preceding (250 epochs). In both cases, a dramatic decrease in accuracy is observed. This is possibly due to the fact that the architecture becomes too complex, leading to overfitting due to over-training.

In general, results indicate that the performance of the normal neural network is better than the pretrained stacked autoencoders network (deep net), except if the number of neurons in each encoder layer is significantly increased. However, once the deep net is fully trained by backpropagation of the entire stack (fine-tuning stage) it outperforms the neural network.

This is attributed to the fact that the model is run based on a train-test split, so it stops training when needed (to avoid overfitting), thus optimizing the use of all the encoders in conjunction.

3 Convolutional Neural Networks

3.1 AlexNet Architecture

In this section, the architecture of AlexNet is analyzed, based on the example provided by CNNex.m.

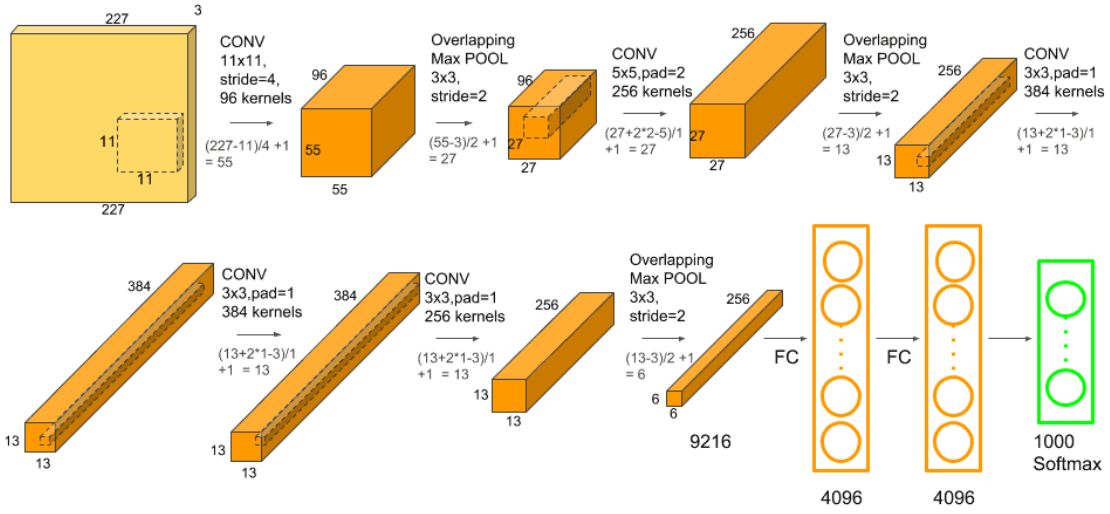


Figure 14: AlexNet Convolutional Neural Network Architecture

Net structure Layer1-Layer2	Parameters	Accuracy (%)	
		Pretrained Deep network	Fine-Tuned Deep Network
1) 100 - 50 neur.	250 epochs L2 regularization: layer 1: 0.004 — layer 2: 0.002 Sparsity regularization: 4 — proportion: 0.15	86.87	99.85
2) 100 - 10 neur.		48.60	99.85
3) 200 - 50 neur.		87.40	99.85
4) 500 - 100 neur.		96.36	99.78
5) 500 - 50 neur.		88.94	99.83
6) 100 - 50 neur.	L2 regularization: layer 1: 0.003 — layer 2: 0.003 Sparsity regularization: 8 — proportion: 0.1	23.95	98.94
7) 100 - 50 neur.	L2 regularization: layer 1: 0.004 — layer 2: 0.002 Sparsity regularization: 8 — proportion: 0.15	80.34	99.81
8) 100 - 50 neur.	L2 regularization: layer 1: 0.001 — layer 2: 0.001 Sparsity regularization: 4 — proportion: 0.15	86.33	99.86
9) 100 - 50 neur.	L2 regularization: layer 1: 0.00001 — layer 2: 0.003 Sparsity regularization: 4 — proportion: 0.15	83.76	99.82
10) 500 - 100 neur.	Default regularization — 1000 Epochs	98.48	98.99
Layer1-Layer2-Layer3			
11) 100 - 50 - 10 neur.	L2 regularization: layer 1: 0.004 — layer 2: 0.002 Sparsity regularization: 4 — proportion: 0.15	29.44	99.70
12) 200 - 100 - 50 neur.	L2 regularization: layer 1: 0.004 — layer 2: 0.002 Sparsity regularization: 4 — proportion: 0.15	75.35	99.84

Normal neural net (1 hidden layer)	1000 epochs training function : trainscg (scaled conjugate gradient backpropagation) performance function: crossentropy	
100 neurons		96.61
500 neurons		95.69
Normal neural net (2 hidden layers)		
100 - 50 neurons		96.60
500 - 100 neurons		95.48

Table 4: Digit classification accuracy for different parameters and architectures. The first 5 experiments explore how the number of neurons in each of the two autoencoders affect accuracy. Experiments 6)−9) show how the regularization parameters affect performance. Note: The type of autoencoder that are trained in the experiments are a sparse autoencoders. This type of autoencoder uses regularizers to learn a sparse representation of the input data.

Weights of first convolutional layer: Each weight is a cell value in a kernel. Each kernel is 3-Dimensional (11x11x3), since it has to be applied to the 3 input channels. In general, the total number of parameters to be optimized in a convolutional layer is equal to: $[h * W * c + 1] * N_f$, where h, w, c are the dimensions of the kernel, 1 is the bias factor and N_f is the number of filters. Therefore, each kernel contains $Height_{kernel} * Width_{kernel} * channels = 11 * 11 * 3 = 363$ *weight terms* $+ 1$ *bias term* so 364 terms in total. These weights are optimized during training, so that each kernel represents a feature appearing in the image. Since the total number of kernels that represent the input layer is 96, the total amounts of weights to be optimized is: $364 * 96 = 34848$.

Input dimension at the start of layer 6 (1st fully connected layer): When a convolutional kernel is applied to an image, the output is a new matrix called "feature map". The output size of a Convolutional layer is calculated using the formula: $(Input\ Size - Kernel\ Size + 2Padding)/Stride = Feature\ Map\ Size$

Input Size : dimension of the input of the convolutional layer. **Kernel Size :** Size of the applied convolutional kernel. **Padding :** size of elements to extend the image beyond its border when applying the kernel. Stride : Specified step size of the kernel as it slides through the image.

Note: The same equation can be used in order to calculate the size of the output of a maxpooling layer with only difference that we use the pooling size instead of the kernel size. The feature map size before the fully connected layer 6 is $[6 * 6 * 256]$(the intermediate calculations can be found in 14).

The input dimension was 227x227x3, which corresponds to 154.587 inputs. Thus, after convolutions, poolings and dropouts, we end up with $6 * 6 * 256 = 9216$ units which is the dimension of the inputs before the fully connected layers. This number is of course much lower to what it would have been if we used fully connected layers directly at the input images. The input used for classification is the output of the last fully connected layer before the SoftMax layer, which has 4096 hidden units.

The three primary advantages of CNNs are: **Local Connectivity:** Convolutional neural networks exploit spatially local correlation by enforcing a sparse local connectivity pattern between neurons of adjacent layers. **Parameter Sharing:** Units in the same convolution kernel share some parameters across layers. This further reduces the effective number of parameters, and therefore improves computational efficiency. **Translation Invariance:** Useful in image classification tasks when positional information are redundant.

3.2 CNN on handwritten digits

Results of different architectures are shown in Figure 15. Epochs were kept the same for each run (15 in total). In general:

1. Increasing filter size (especially in small images) is a bad idea since the network is not able to capture local pixel relations.
2. Adding more pooling layers downsamples the features, which decreases accuracy (however this could also prevent overfitting). Conversely, reducing pooling dramatically increases accuracy.
3. Smaller filter sizes and increasing the amount of filters gives better results.
4. Even if architecture is simplified to just one convolutional layer of more (40) and smaller (3x3) filters, accuracy is dramatically increased.

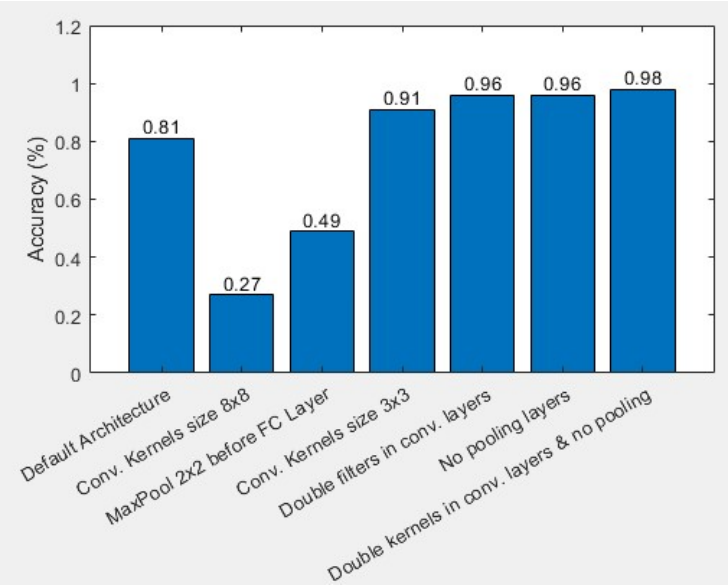


Figure 15: Experiments with different CNN Architectures

CNNs over MLPs: CNN weights are smaller and shared, and layers are sparsely and not fully connected, hence easier to train. Filters take advantage of spatial patterns (e.g. a table), which they can find and match regardless to where the pattern is located in the picture.

Part IV
Generative Models

1 Restricted Boltzmann Machines (RBMs)

An RBM is trained using binary images of handwritten digits (MNIST dataset). Afterwards, using Gibbs sampling on the learned distribution that the RBM represents, reconstructed images of the handwritten digits are reconstructed. The effects of different training parameters such as number of training epochs, number of neurons in the RBM, compared to an original part of the test set can be found in the figure below.

They are summarized as follows:

Epochs and neurons: As expected, increasing the number of epochs and the number of neurons increases the accuracy of the reconstructed image.

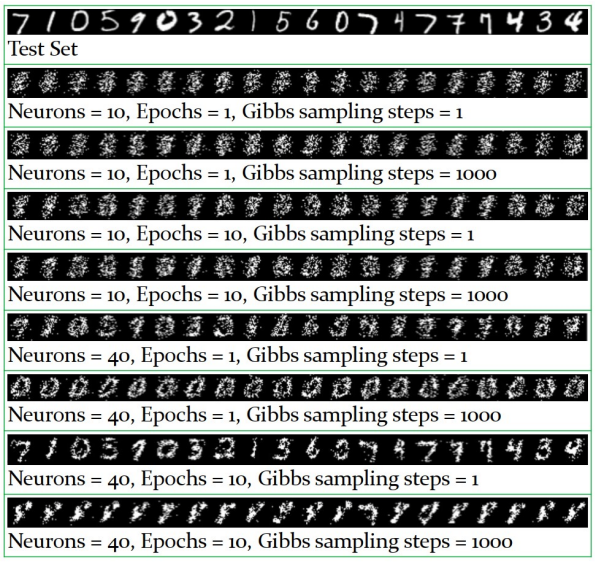


Figure 16: Reconstructed images for several model architectures and Gibbs sampling steps (for learning rate=0.01)

Gibbs steps: Increasing the number of Gibbs steps decreases the accuracy of the reconstructed image. That's because for a given test image (bold denotes a vector of features-pixels), what 1 Gibbs sample does, is that it first calculates:
$p(h_j = 1|\mathbf{v}) = \sigma(b_j + \sum_{i=1}^{m} w_{i,j}u_i)$ meaning that it calculates the probabilities of each hidden unit of hidden units being 1. Then, after assigning the most probable value for each hidden unit, it reconstructs the image (the visible units), by calculating for each visible unit of the vector its conditional probability, that is:
$p(u_i = 1|\mathbf{h}) = \sigma(\alpha_i + \sum_{j=1}^{m} w_{i,j}h_j)$

The calculations above make one Gibbs sample. It is an unbiased sample of the data v. However, it's a very biased sample of the model (towards the data). Therefore, the reconstructed image, due to the data bias, looks very similar to the test data. However, after repeating this sample generation enough times, an unbiased sample of the general model distribution $P(v, h)$ is generated, which is why after a lot Gibbs sampling rounds, all the reconstructed images look very similar. Due to the simple architecture of the RBM, the underlying distribution, as observed in the experiments, cannot represent all the digits, converging to a low energy equilibrium far from the ideal.

Reconstruction of images' missing parts:
As stated above, increasing the number of neurons and epochs improves the reconstruction quality. Gibbs steps are kept low in order for the samples to be biased towards the training data. The learning rate suitability depends on each problem: too low and the solution may take long time to converge or get trapped in local optima, too high and the solution may not find a 'good' optimum. In Figure 17, the reconstruction of some test images after one Gibbs sampling step are shown for different learning rates. It appears that an ideal value of learning rate is about 0.05-0.1. Therefore, a network of 100 neurons, 0.05 learn rate after 10 iterations for 1 Gibbs sample was chosen to check the ability of the network to reconstruct images after removal of different rows.

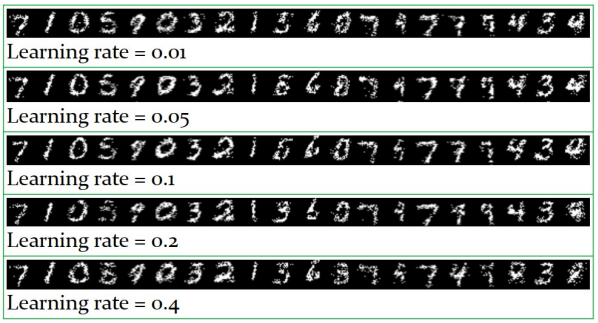
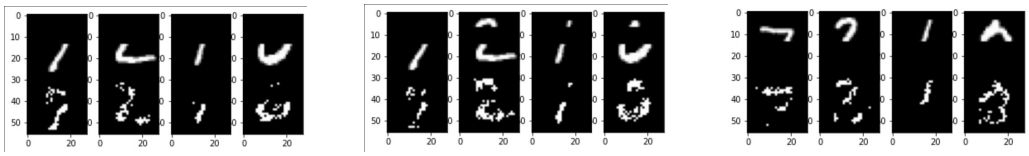


Figure 17: Effect of learning rate for an RBM of 40 neurons after 10 iterations (1 Gibbs sampling step)

In general, the reconstruction quality depends on the part of the digit that was removed and how unique that part was for the digit. For example, removing the top part of the digit 2 (rows 1:14) leads to a good reconstruction while removing the bottom part (14:28) leads to a 'bad' reconstruction, since the model confuses 2 to be a 3.



(a) Removing rows 1-14 (b) Removing rows 7-15 (c) Removing rows 14-28

Figure 18: Reconstruction of the digits 7,2,1,0,4 of the test set, after removal of their top, middle and bottom part respectively

2 Deep Boltzmann Machines

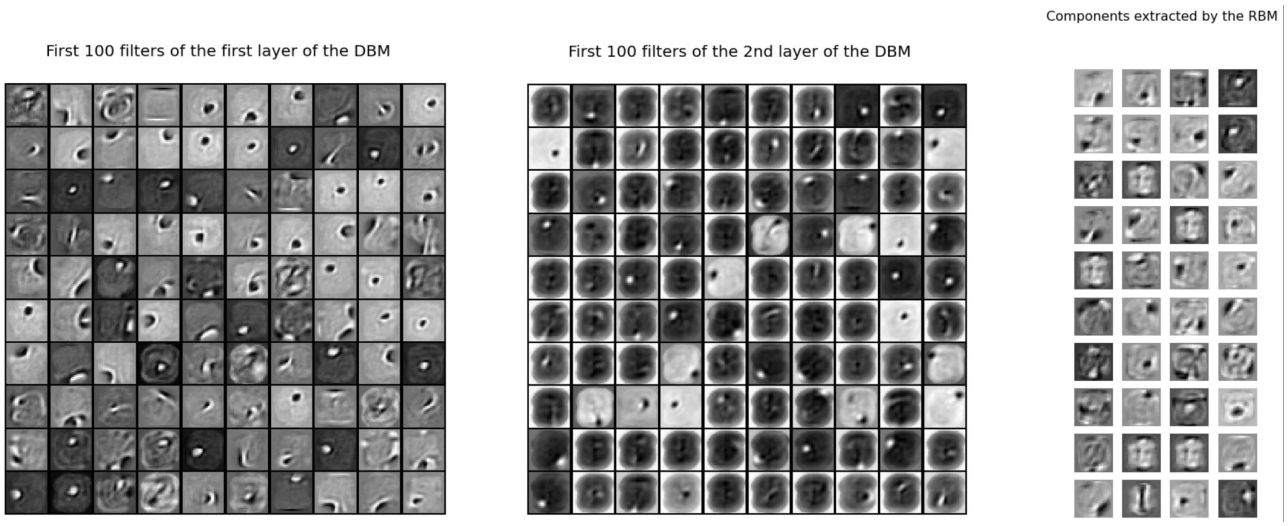


Figure 19: Filters from the DBM and RBM layers

Filters:
The RBM filters from exercise 2.1 (with architecture: 40 neurons, 10 iterations, 1 Gibbs sampling step and 0.05 learning rate), as well as the 1st and 2nd layers' filters of the DBM are shown above. This filter visualization is equivalent to representing the weights of each hidden layer as values in a gray-scale image. This way, we can see which features "excite" each hidden layer. Apparently, both the RBM's filters and the 2nd DBM layer's filters visualizations look similar and abstractly resemble handwritten digits. On the other hand, the 1st DBM layer's filters is less similar to the RBM filters. This can be attributed to the fact that in the first case (2nd DBM & RBM), the layers learn to represent the high level features (general shape) of the numbers. In contrast, the 1st DBM layer represents lower level features such as edges, splines etc., which are then combined in the next layer to extract the more general representation.

Sampling quality:
Image samples from the pretrained DBM's distribution are shown to the right. By comparing these to the RBM's learned distribution in Figure 16, a considerable increase in image quality is observed in the DBM case. DBMs are constructed by sequentially training each RBM layer in greedy manner, then stacking them together and perform fine-tuning via back propagation. Therefore, much like a multi-layer neural network, the DBM is able to model more complex representations than a 1-hidden layer network (see XOR problem). A DBM can have improved performance over a simpler RBM.

Figure 20: Sampling from the distribution learned by the DBM

3 Generative Adversarial Networks

A sample of the training data (images of a single class), as well as generated images after 1000, 7000 and 20000 batches

are shown in the figure below. The increase in image quality is reflected on the performance curves. By inspecting these curves, the following observations are made:

- After a point, the discriminator accuracy hovers around 50%, which means that the discriminator has low confidence in whether the images it receives are real or not.
- Around 5-10k batches are needed to achieve optimal results.
- After 12.5k batches, a decrease in generator accuracy and increase in discriminator accuracy is observed. This can be attributed to the fact that after this point the model stability gets compromised, which eventually leads to mode collapse. If a generator produces an especially plausible output, it may learn to produce only that output (since it is always trying to find the one output that seems most plausible to the discriminator). If the generator starts producing the same output (or a small set of outputs) over and over again, the discriminator's best strategy is to learn to always reject that output. That can possibly explain the increase in discriminator's accuracy observed in the graph. However, if in the next iterations the discriminator gets stuck in a local minimum and doesn't find the best strategy, then it's trivial for the next generator iteration to find the most plausible output for the current discriminator. Each iteration of the generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the local minimum. As a result the generators rotate through a small set of output types.

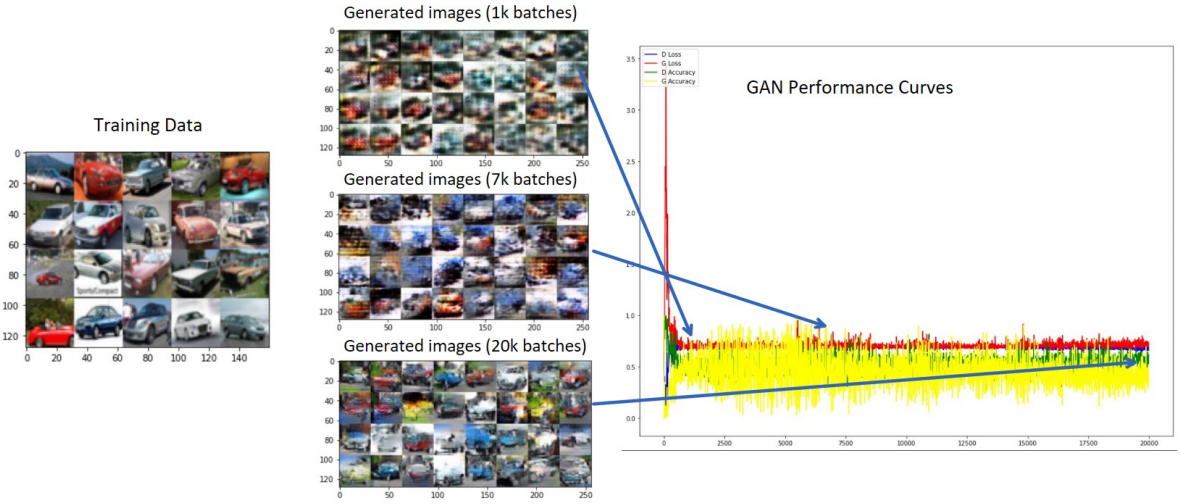


Figure 21: GAN Training Samples, generated samples & performance curves

4 Optimal Transport

OT: The goal of performing optimal transport is to transform the pixel RGB histograms of one image to match the histograms of the other image, hence enabling both images to share the same probability distribution of their pixels. However, using optimal transport, the following concepts are optimized:

- The total amount of color intensity (normalized value in the 0-1 range of the RGB channels for each pixel) is the "sand" to be optimally rearranged.
- The squared Euclidean distance between vectors of pixel coordinates is the 'distance' that this dirt will be re-arranged in order for the probability distributions to match.
- By applying regularization, the total amount of color intensity distributed becomes more balanced between each pixel, hence they have less "pronounced" intensities compared to their non-regularized counterparts.

The results of this domain adaption for two random (same-sized) landscape images are shown in the figure below :

Figure 22: Swapping the probability distribution of pixel RGB intensity of two images

Non-optimal colour swapping would most likely result in artifacts appearing in the transformed image, and a general un-natural and non-consistent alteration, instead of the smooth transition observed here. Just swapping the pixels would simply mean that one image gets transformed to the other.

WGAN: JS divergence is the distance metric used in GANs, and has gradient issues leading to unstable training. Instead, WGAN bases its loss from Wassertein Distance between the distributions of the real vs generated data. In the figure below, the performance curves as well as generated samples for different batch sizes are shown. Apparently, the WGAN already at 2.5k iterations generates more realistic-looking digits than its standard counterpart (loss metric is different hence the big discrepancy in performance curves).

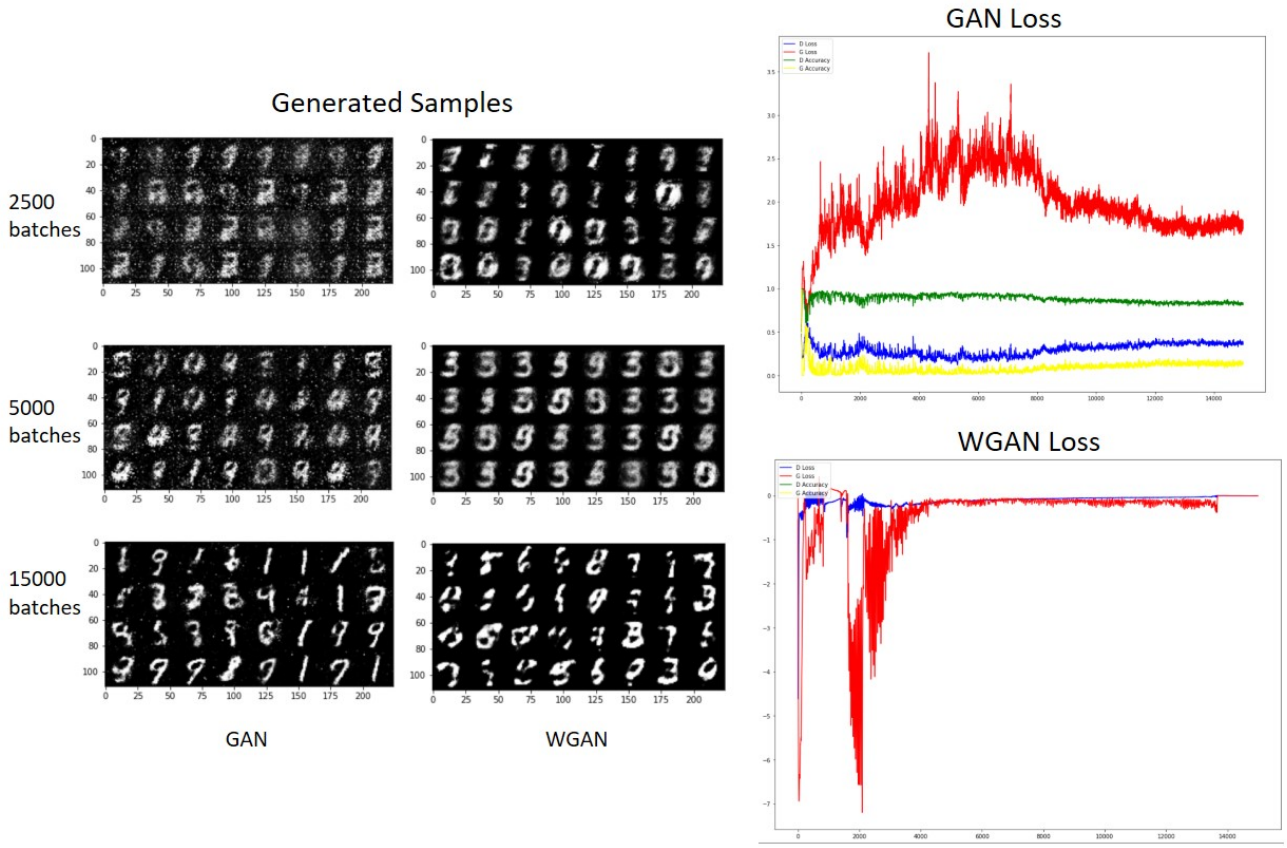


Figure 23: Performance comparison between a simple GAN and WGAN