Artificial Neural Networks and Deep Learning - Assignments report

Deniz Soysal - r0875700

June 1, 2022

1 Assignment 1 : Supervised learning and generalization

1.1 Backpropagation in feedforward multi-layer networks

1.1.1 Function approximation: comparison of various algorithms

For this part, we will compare the performance of different algorithms for approximating a sine function. The algorithms we compare in Matlab are traingd (gradient descent), 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) and trainlm (Levenberg-Marquardt algorithm). The function we want to approximate is $y = sin(x^2)$. We want to approximate y for $x : [0, 3\pi]$. Therefore, the input variable is x, while the target variable is y. The step size for the input variable x is 0.05 (so we take $\frac{3\pi}{0.05}$ data points).

The network we train is a MLP with a single hidden layer containing 50 neurons. The results are depicted at Figure 1. We can observe a similar behaviour between each algorithm : the network is more accurate to estimate the value of y for low values of x (for $x = [0...4]$). For higher values of x, the estimation of the network becomes less accurate. This can be explained easily by our target function $y = sin(x^2)$ and the **constant** sample step we take for x, i.e. 0.05. For low values of x, the change of y is low when increasing x by 0.05, while for high values of x, y changes much more when increasing x by 0.05. Thus, this can be seen as the network receiving more data to estimate the slope of the curve for low values of x. This can be observed at Figure 1 : even if the step size between each input sample (represented by the blue 'X' at the Figure 1) is the same, it seems like they are much closer for low values of x.

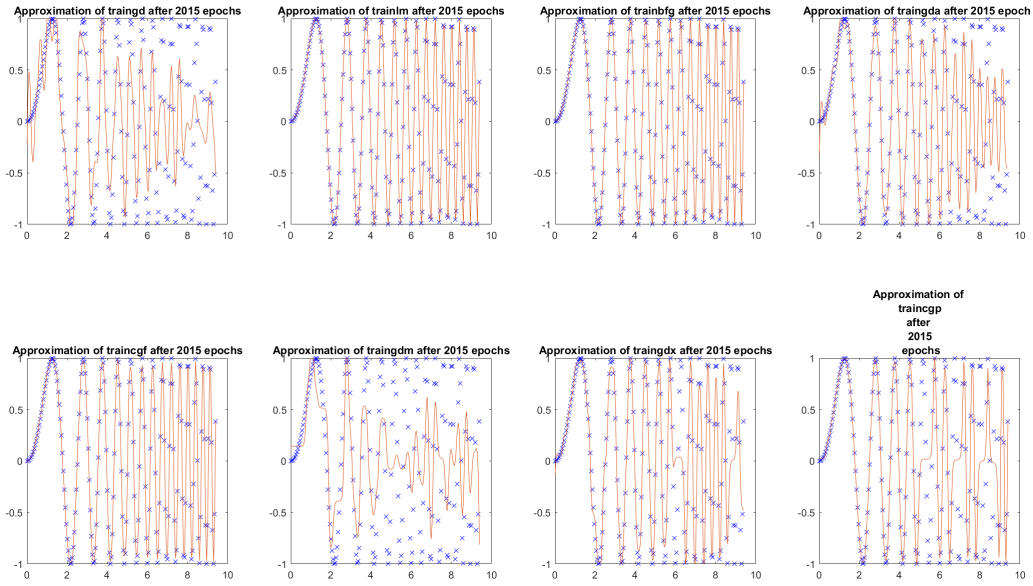


Figure 1: Approximation of the function after 2015 epochs

We can also observe that the Levenberg-Marquardt algorithm (trainlm at Figure 1) , the Quasi Newton algorithm (trainbgf at Figure 1) and the Conjugate Gradient algorithm (traincgf at Figure 1) performs well, while for other cases the network has some difficulties to approximate the function. Let's see how the MSE evolves against the number of epochs for each of the algorithms. The results are displayed at Figure 2. It becomes clear that the trainlm algorithm is the best one in term of MSE. In fact, we can separate the curves to four groups, going from the top to the bottom at Figure 2 :

1. *traingd,traingdm,traingda,traingdx* : Gradient-based methods.
    - We can see at Figure 2 that $MSE_{traingdm} < MSE_{traingd}$, therefore having a momentum term helps a little bit. The idea of the momentum term is as follows : in the "standart" gradient descent, the update of the weights at time t_i depends on the gradient at time t_i : we do not take into account the previous

gradient values. So when we reach a local minima, we can get easily stuck. Momentum tackle this problem by introducing a term proportional to a moving weighted[1] average of the previous gradients.

- $MSE_{traingda}$ is even better, with the use of an adaptative learning rate which reduces when the loss value reach a plateau. $MSE_{traingdx}$, which is the combination of a momentum term and an adaptative learning rate gives the best result for the gradient descent algorithms. However, for each of these four methods, we get rapidly stuck in a local minima, and the MSE stays very high compared to the following methods.

2. *traincgf,traincgp* : Conjuguate-gradient methods

   - At Figure 2, we can see that the performance of these methods are better compared to the Gradient-based methods. The idea is quite similar to using a momentum term : we also use previous gradient values to update the weights. The difference is that we do not consider an average of previous gradient, but the current gradient must be orthogonal to the previous steps. This leads, especially for quadratic functions, to an algorithm that converge faster than a standart gradient descent algorithm.
   - Another big advantage of the Conjuguate-gradient method, compared to Newton methods is that it uses only vectors to update the weights, and not matrices.

3. *trainbfg* : Quasi-Newton method

   - We can see at Figure 2 that $MSE_{trainbfg}$ is better than Gradient-based and Conjuguate-gradient based methods : after only 0.5 sec, the network has already reached a MSE that is far less than the previous methods (even if the training continues afterwards). The idea of the Newton Algorithm is to also make use of the Hessian matrix when training the network[2]. The Hessian Matrix being a second order derivative matrix, it can be difficult to construct. Quasi-Newton Algorithms is a way of simplifying the updates by iteratively approximating the Hessian Matrix with only first order derivatives.

4. *trainlm* : Levenberg-Marquardt method

   - In the Newton Algorithm, the step size Δx is based on the inverse of the Hessian Matrix. If this matrix is ill conditionned, we cannot compute Δx. Moreover, there is no constraint on Δx, which can lead to a large Δx. The idea of the Levenberg-Marquardt method is to impose a constraint on Δx, namely $||\Delta x||_2 = 1$. We obtain a step size $\Delta x = -[H + \lambda I]^{-1} g$. The term λI makes sure that all values of the matrix we inverse, i.e. $H + \lambda I$ are positive and thus handles the ill-conditionned case[3]. λ is the Lagrangian multiplier. When $\lambda = 0$, we obtain the same step as in a Newton Method. On the other hand, when $\lambda I >>$, the term g will be dominant (g being the gradient) so the update will be similar to a steepest descent algorithm. In **Matlab**, the initial value[4] is $\lambda = 0.001$ (so we start with a step similar to Netwon Method's) and the value of λ is increased through the epochs. This leads to a very performing and fast converging algorithm. We see at Figure 2 that $MSE_{trainlm}$ outperforms the other methods.

1.1.2 Learning from noisy data: generalization

Here, we compare the performance of the same algorithms but to approximate a noisy version of the function $y = sin(x^2)$. The added noise is a Gaussian noise with $\sigma = 0.2$. At Figure 3a, we can see that in presence of noise the performance drops[5] for almost all algorithms. In fact, what happens here is overfitting : as we can see at Figure 3b where we compare the approximation of the network for noisy data using trainlm after 40 epochs and after 2000 epochs, the approximation of the sine function is better after 40 epochs ! After 2000 epochs, the network start to *learn the noise*, and thus generalize less. This phenomena is called **overfitting**, and correspond to the case where the network has a very high performance on the training data but fails to generalizes well. The causes of overfitting can be a too complex model, or a model trained for too much epochs. Regularization methods exists to avoid overfitting.

[1] The weights are defined in a exponential manner, with the most recent gradient having the higher value

[2] The Hessian matrix appears in the optimization equation when we express the loss function as a 2 terms Taylor Expansion. The optimal step of the weights is found by minimizing the loss expressed in this form.

[3] namely when a value of the matrix is zero. In these cases inversing the matrix becomes impossible

[4] according to https://nl.mathworks.com/help/deeplearning/ref/trainlm.html

[5] i.e. the MSE is higher

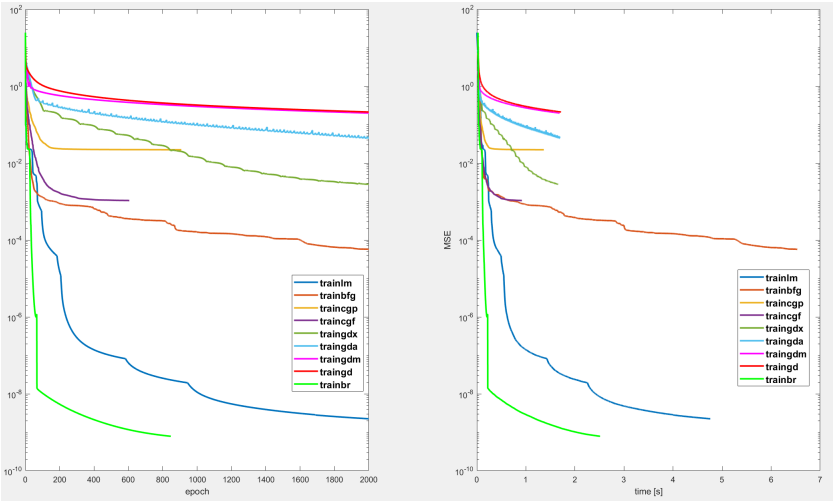
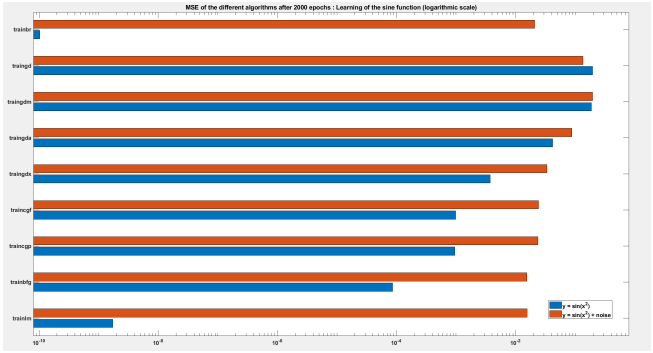
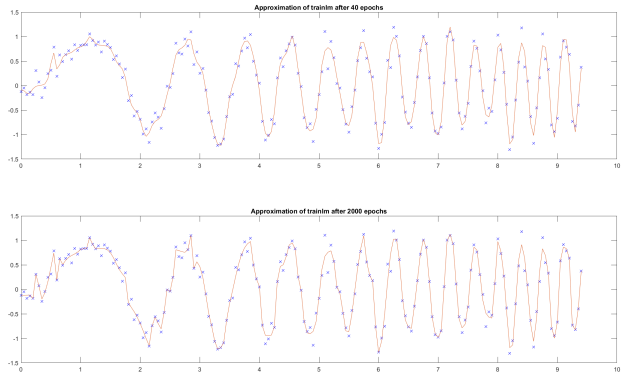


Figure 2: MSE by epoch and MSE by time



(a) Performance in terms of MSE of the networks, on noiseless (blue) and noisy data (red)



(b) Regression using trainlm, after 40 epochs (top figure) and 2000 epochs (bottom figure).

Figure 3: Left Figure : Performance of the networks. Right Figure : Overfitting phenomena

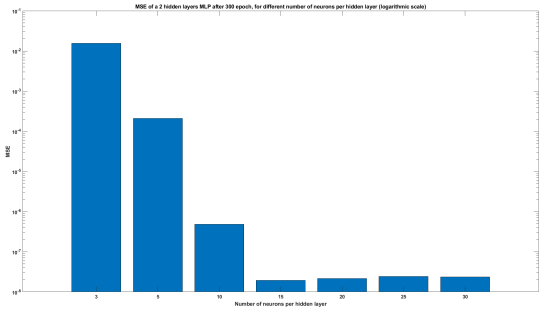
1.1.3 Personal Regression Example

For this part, we will try to learn a target value t based on two inputs x_1 and x_2. We will use a dataset of 13600 samples, from which we will use 3000 samples divided into three sets of 1000 samples: training, validation and test set. It is important to sample *without replacement* to avoid having the same sample on different sets[6].

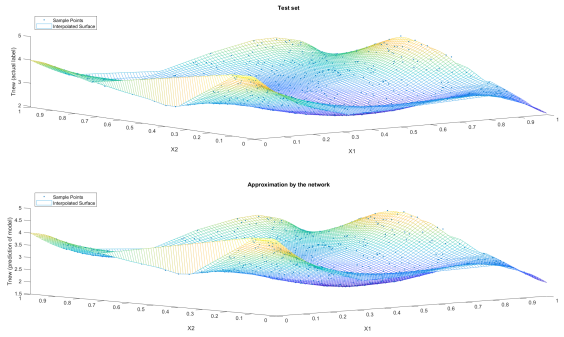
The use of a validation set helps against overfitting : Matlab implement an early stopping method, which stop the training when the validation increases during "max_fail" epochs (default value for max_fail is 6). The algorithm we will use is *trainlm*, which has proven in the previous section to be very powerful. We will run the network for a maximum number of 200 epochs.

The results are displayed at Figure 4a. As we can see, the best performance is achieved for an MLP with 15 neurons in each hidden layer. After that, the MSE increases, showing that a too complex neural network can lead to worse performance due to overfitting. To improve the performance of the network, two ideas to explore are : first, use more training samples. Second, use regularization to avoid overfitting. We will talk about regularization in the next section. At Figure 4b, we can observe the test set and the estimation of the network, and conclude that the network is able to estimate the output value of the test set.

[6]if some samples of the test set already appears in the training set, the test performance will not be truly representative of the performance of the network on unseen data



(a) MSE vs nb of neurons per hidden layer (MLP with two hidden layers)



(b) Top : test set, bottom : approximation by the network

Figure 4: Left Figure : MSE vs nb of neurons. Right Figure : Approximation of the test set

1.2 Bayesian Inference

The *trainbr* algorithm is a regularized version of the *trainlm* algorithm by adding a weight decay term in the objective function. This helps the model to avoid being too complex by minimizing the cost function and the effective number of the parameters at the same time. Let's compare the performance of the trainbr and the trainlm algorithm to see how regularization can improve the performance. Let's reconsider the case where we try to approximate $y = sin(x^2)$ and $y = sin(x^2) + noise$ using neural networks. As we can see in Figure 2, the trainbr algorithm outperform the trainlm algorithm and is the one that performs the best.

Let's also compare trainlm and trainbr on highly overparametrised network. To do so, we will consider the personal regression example of section 1.1.3. We use a overparametrised network with 3 hidden layers of 40 neurons each. We compare trainlm and trainbr by running the algorithms for 1000 epochs. We monitor the validation loss and stop the algorithm if we reach a plateau. The MSE on the test set is presented at Table 1. As we can see, trainbr, by allowing regularization, allows higher performance on overparametrized networks.

Algorithm	MSE on test
trainlm	1.92118e-08
trainbr	1.60809e-10

Table 1: Comparison of the performance of trainlm and trainbr on a overparametrized network

2 Assignment 2 : Recurrent neural networks

2.1 Hopfield Network

2.1.1 Hopfield Networks with two neurons

We create a Hopfield Network with three attractors :

$$T = \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & -1 \end{bmatrix} \tag{1}$$

We will analyze the dynamics of this network by looking to the activity of the network for different inputs. The inputs values for the two neurons will be random values [randn,randn], values at the origin [0,0], values on the x-axis [randn,0] and values on the y-axis [0,randn].

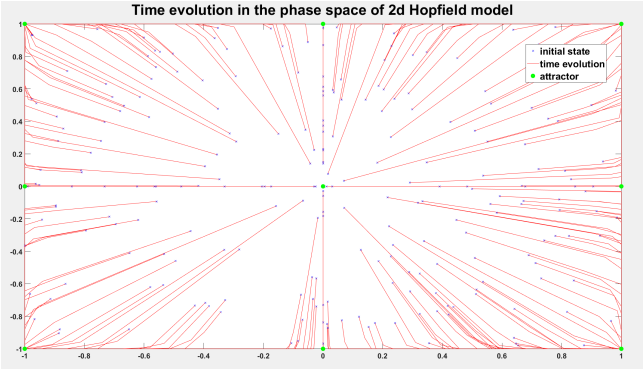


Figure 5: Activity of the two neurons Hopfield Network for different inputs

As we can see at Figure 5, even if the Hopfield Network was initialized with three attractors as shown at Equation (1), the network has much more equilibrium points, which we can identify as :

$$\begin{bmatrix} -1 & 0 & 0 & 0 & 1 & -1 \\ 1 & 0 & 1 & -1 & 0 & 0 \end{bmatrix}$$

These are spurious states, or in other terms unwanted equilibrium points of the dynamical system. When storing a pattern ξ^ν, $-\xi^\nu$ also becomes an equilibrium point[7] : that explains why [-1 1] is an attractor. We can also remark that we have attractors at points where one of the neuron (or both) is at a zero state. The number of epoch to reach an equilibrium points depends on the initial values, but we have noticed that for less than 30 epochs, we can reach an equilibrium points from almost all inputs.

2.1.2 Hopfield Networks with three neurons

This time, we initialise a three neurons Hopfield Networks with :

$$T = \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix} \tag{2}$$

Therefore, we have a three neurons network and we initialize it with three attractors. As we can see at Figure 6, we again have spurious states, even if they are more complicated to discover in a 3D space. Moreover, the network takes more iteration to converge towards an equilibrium point compared to the two neurons case.

2.1.3 Handwritten digits as attractors

Here, we try to reconstruct noisy digits with Hopfield Networks. The noise is Gaussian, added to the maps of the digits. As we can observe at Figure 7, even if the noise makes the recognition of the digits impossible for humans, the Hopfield Network is able to converge to the right numbers ! Even after two iterations, the numbers are already recognizable. After 30 iterations, it is hard to tell the difference between the attractors and the reconstructed images.

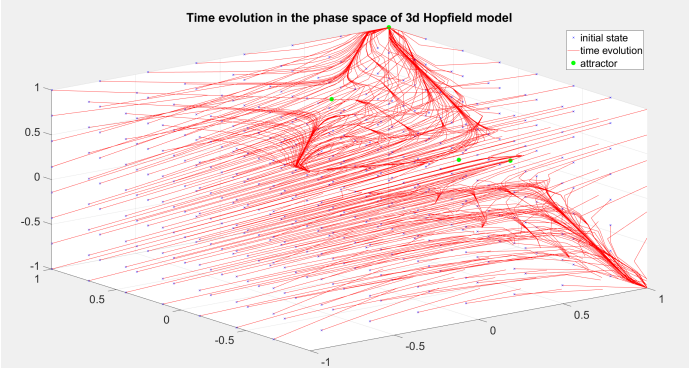


Figure 6: Activity of the three neurons Hopfield Network for different inputs

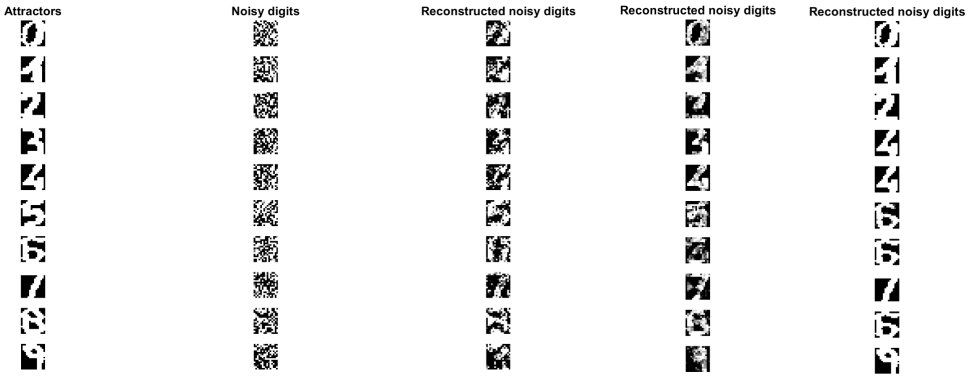
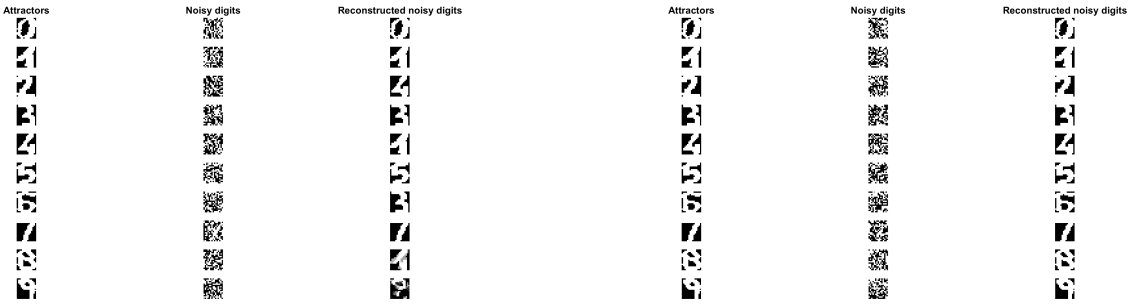


Figure 7: Attractors, noisy digits, and reconstructed noisy digits after 1, 2 and 30 iterations (noiselevel = 5)



(a) Reconstruction after 30 epochs, noise = 6

(b) Reconstruction after 50 epochs, noise = 6

Figure 8: Reconstruction after 30 and after 50 epochs when noiselevel = 6

As we can observe at Figure 8, when the level of the noise increase, the reconstruction becomes more difficult : at 30 epochs, some of the neurons have wrong values. For example, for the "2" input, the reconstruction is "4". However, after 50 epochs, the values are corrected, which proves the power of such networks. However, when the noise level is too high (noiselevel > 10), the network converge to the wrong numbers.

2.2 Long Short-Term Memory Networks

The goal here is to predict the 100 next data points of the Santa Fe dataset based on the first 1000 data points. What we do is therefore time-series prediction. To do so, we train the model on the first 1000 data points, and use this trained model to predict the next 100 data points. The data is one dimensional. The model is trained as

[7] in the case there is no bias term in the model

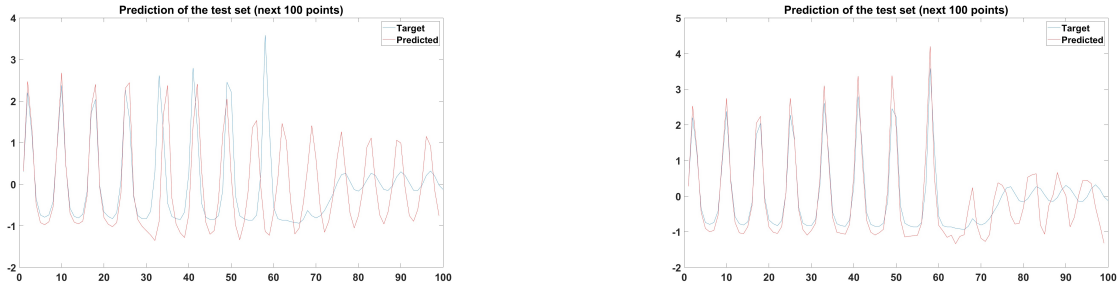
follows : we divide the dataset into pairs of input and target, where the input is a window of data of size p, and the target is the next point, i.e. point $p+1$. We shift the window accross the whole x-axis to cover the whole dataset. As an example, consider a time series in the form [1;2;3;4;5]. If we select a window size of 2, the pairs of (input;target) will be [(1,2;3),(2,3;4),(3,4;5)]. After training the model, to predict what is the next unknown value, we place the window at the end of the dataset and predict the next point, i.e. in our example : : (4,5;next point). After, we can use the predicted next point to predict even further, i.e. in our example :(5,next point;even further).

2.2.1 MLP

Here, we will use a one hidden layer MLP to predict the next 100 points. Note that we run the algorithm several times to avoid bad initialisation of the weights. To select the number of hidden units and the window size p, we will try several values and compare the MSE. The training is performed using the *trainlm* algorithm.

MSE	P=10	P=20	P=30	P=40	P=50
H=10	1.6669	1.5141	1.8983	0.2145	0.2235
H=50	2.0020	1.5613	0.8523	0.2248	0.1826

Table 2: MSE for the predicted 100 values for varying window size and number of hidden units



(a) 10 hidden units, window size of 10

(b) 50 hidden units, window size of 50

Figure 9: Prediction of the next 100 data points using MLP

As we can see at Table 2, the best MSE we have (0.1826) is obtained when using 50 hidden units with a window size p of 50. This is also observable at Figure 9b, where the prediction when using 50 hidden units with a window size p of 50 is closer to the target compared to Figure 9a.

2.2.2 LSTM

LSTMs answer the vanishing gradient problems of RNN by also being able to learn long-term dependencies. The idea of LSTMs is to write, read and erase information from the canal (in red at Figure 10) via gates. The first gate receives input information from the previous neuron and a sigmoid activation. The second gate also receives information from the previous neuron but has 2 activation function : sigmoid an tanh. The output with a sgmoid activation is equal or lower to the input : the sigmoid activation implies a "forget gate", i.e. we erase information from the canal. With the tanh activation, we add new information to the canal. The third gate monitor the quantity of information to send to the output of the cell, h_t. at Figure 10

In RNN, the problem of the vanishing gradient is caused by backpropagating the error through a path where we multiply several times by an activation function with a derivative lower than 1 (blue path at Figure 10). In LSTMs, we have a path for backpropagating the error without any multiplication with an activation function (red path in Figure 10) : this solves the problem of vanishing gradient and allow to store long term dependencies ! The short term memory in LSTM correspond to the yellow path in Figure 10.

Based on the *TimeSeriesForecastingUsingDeepLearningExample*[8] example of Matlab, we have implemented an LSTM prediction for the Santa Fe dataset to compare it with the MLP solution. We have tried for several number of hidden units and window size. The optimizer used is *adam*. The results are presented at Table 3. We can see that the best performance was when using 50 hidden units with a window size of 50. The MSE obtained (0.0874) is much lower compared to Table 2 where we use a MLP. This proves that having long term dependencies combined with short-term dependencies allows the network to learn more precisely how will the function evolves over time. At Figure 11b, we can see that the LSTM is able to predict accurately the evolution of the function.

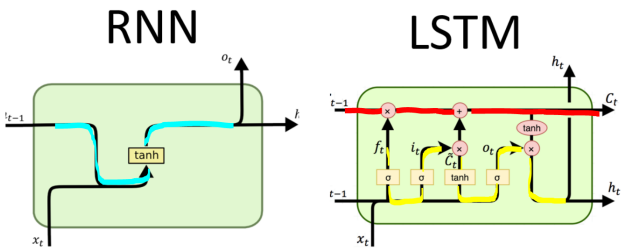
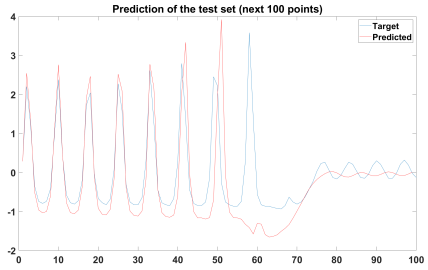


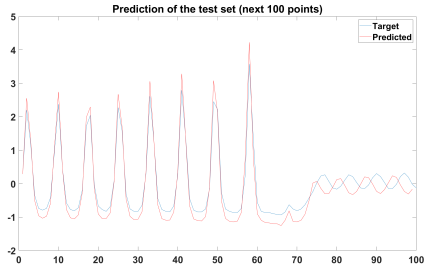
Figure 10: RNN vs LSTM

MSE	P=10	P=50
H=10	0.5074	0.1009
H=50	0.8146	0.0874
H=100	0.5801	0.0981

Table 3: MSE for the predicted 100 values for varying window size and number of hidden units



(a) 10 hidden units, window size of 10



(b) 50 hidden units, window size of 50

Figure 11: Prediction of the next 100 data points using LSTM

[8] that we can open with the command openExample('nnet/TimeSeriesForecastingUsingDeepLearningExample')

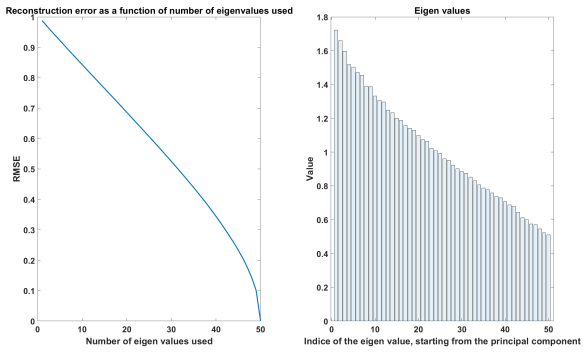
3 Assignment 3 : Deep feature learning

3.1 Principal Component Analysis

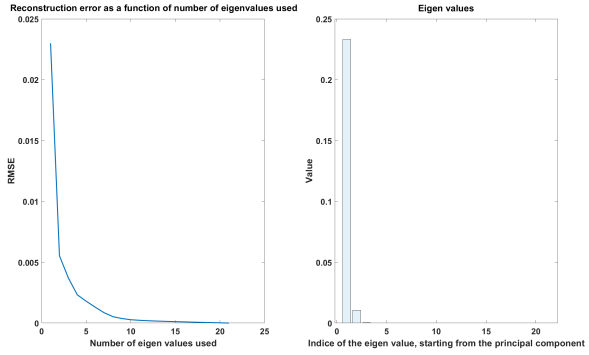
The idea of Principal component analysis is to find the direction of maximal variance, by computing the largest eigenvectors of the covariance matrix of the data distribution. Large eigen values correspond to components that are the most informative for the data distribution, while very small eigen values are usually linked to noise. PCA is used to do dimensionality reduction. Consider a dataset of dimension N. By projecting the data to the M principal components, we can reduce the dimensionality from N to M.

3.1.1 Redundancy and Random Data

We generate a dataset of random Gaussian numbers. The dataset has a dimension of 50. As we can observe at the right plot of Figure 12a, it is difficult to choose a number of eigen values to keep that will still explain accurately the data : the eigen values decreases linearly, and so is the RMSE at the left plot of Figure 12a. This is expected because the error is proportional to the sum of all the eigen values we ignore. The fact that there is not really eigen values more important than the others can be explained by the nature of the data : it is a dataset of random Gaussian numbers, so there is not a direction on which the variance is a lot bigger compared to other directions.



(a) Reconstruction error and eigen values for the random numbers



(b) Reconstruction error and eigen values for choles_all dataset

Figure 12: Reconstruction errror and eigen values for random numbers and choles_all dataset

On the other hand, at Figure 12b for the choles_all dataset, we have a different distribution : the first eigen value is the most important one. By just using the first eigenvector to represent the data, the reconstruction error, in terms of RMSE is already less than 0.03. If we use the first and the second eigenvector, the reconstruction error is around 0.005.

3.1.2 Principal Component Analysis on Handwritten Digits

Now, let's apply PCA to handwritten digits. We have a dataset of 16x16 images (the dimensionality of the data is therefore 16x16 = 256). The mean image of the dataset is displayed at Figure 13

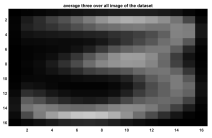
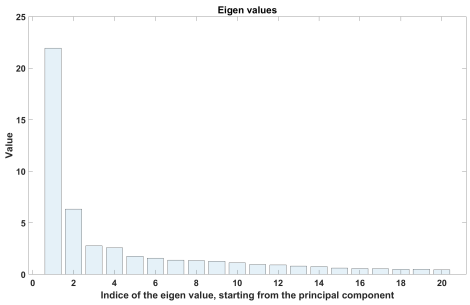


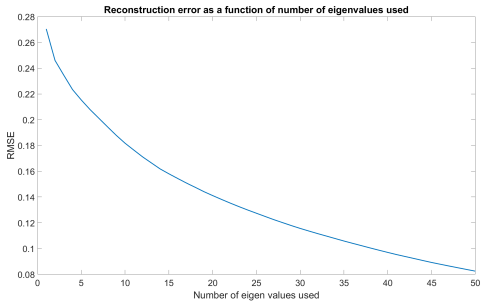
Figure 13: Mean image of the dataset

If we look at the eigen values, we can observe at Figure 14a that the first eigenvalue is much more important than the other ones. Just using this one must be enough to have a fair representation of the data. We can see at Figure 14b that the reconstruction error is around 0.27 when using only the first eigen value. We can also observe that the reconstruction error goes down steeply if we add the second eigen value to be around 0.24, but then the Reconstruction Error decreases more slowly by adding more eigen values. This again can be explained by the fact

that the error is proportional to the sum of all the eigen values we ignore, and the first 2 eigen values are the most important ones.



(a) Eigen values



(b) Reconstruction error vs # of eigen values

Figure 14: Eigen values and reconstruction error

When we define q=256 (i.e. use all dimensions of the data), we except to have an error of 0 between the reconstructed image and the original one. However, we obtain an error 4.7227e-14. This can be explained by truncation errors of the matlab functions.

3.2 Stacked Autoencoder

An **autoencoder** is a neural network that allows compressing data representations. The input layer and the output layer have the same dimension which is the dimension of the input vector. The hidden layer has a smaller dimension than the input vector. The objective function is to minimize the difference between the input vector and the output vector. After training, we can use the network to build a compressed representation of the data : indeed, the network will learn to build a hidden representation informative enough to be able to re-build at the output layer a vector close to the input vector ! To have a compressed representation of the digits, we impose a sparsity constraint by imposing that the dimension of the hidden layer of the autoencoder must be smaller that the dimension of the input. Note also that the dimension reduction depends on the problem : some data can be explained with a very small feature vector, while other cannot be too much compressed. Here, we will try to see the effect of compression by looking at different hidden layer size (especially looking at the effect of the hidden layer size of the first autoencoder which compress the original data).

A **stacked autoencoder** is the idea to use multiple autoencoders in the following way : we train a first autoencoder to reduce the dimensionality from D to z. Based on z, we use a second autoencoder to reduce even more the dimensionality to m, etc. At the end, we use the compressed representation of the last autoencoder to train a classifier. The autoencoders are trained in an unsupervised way, while the classification layer is trained in a supervised way. Finally, we merge all layers to fine tune the network.

The dataset we use is a synthetic dataset of 28x28 pixels images of digits from 0 to 9. Thus, we have 10 classes, and the input feature vector is of size 28x28 = 784. We will train a stacked autoencoder to compress the input representation, and to finally train a classification layer.

3.2.1 Results

At Table 4, we can observe that the best performance is obtained after fine-tuning the network with architecture [100 50], therefore 100 neurons in the first network and 50 neurons in the second one. We can observe that the network with architecture [20 10] performs poorly : indeed, when compressing the data onto a dimension of 20 with the first autoencoder, we will loose too much information. We can also observe that fine-tunning helps to increase the performance.

Accuracy [%]	H=[20 10]	H=[50 25]	H=[100 50]	H=[200 100]
Before fine-tunning	17.7	48.4	85.1	97.7
After fine-tunning	59.4	99.6	99.8	98.6

Table 4: Accuracy of the stacked autonencoder on the test set, before and after fine tunning for different layers size

3.2.2 Comparison with mlp

When using MLP, the performance is lower compared to the use of autoencoders. This can be observed at Table 5.

Accuracy [%]	H=[20 10]	H=[50 25]	H=[100 50]	H=[200 100]
	89.7	94.0	97.1	97.7

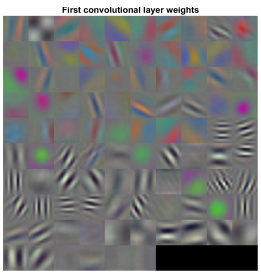
Table 5: Accuracy of MLP for different layers size

3.3 Convolutional Neural Networks

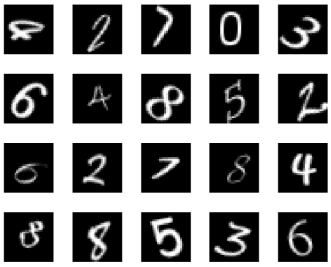
A convolutional neural network is a deep learning architecture composed of convolutional layers, pooling layers and fully connected layers. Each convolutional layer *convolve* filters accross the image to look for patterns. Pooling layers helps for compressing the representation. The filters are learned by the network itself. After having extracted the features thanks to convolutional and pooling layers, we can use fully connected layers and a classification layer to classify the learned features. The first convolutional layers capture basic image features such as edges, while the next convolutional layers learns more complex features such as nose, eyes, etc in a face classification problem.

3.3.1 CNNex.m

Here, we will analyse the network used in the CNNex.m script which classify images using a CNN[9] to extract features, and a SVM to classify the extracted features. The input images have dimension (227,227,3) while the output is 1000 classes. The first convolutional layer has 96 filters of size [11 11]. So, we will build 96 feature maps of the input with 11x11 filters. The weights of the 96 filters are learned by the network. The output of the first convolutional layer is shown at Figure 15a.



(a) Output of the first layer, for the flower data (CNNex.m script)



(b) Handwritten digits to classify (CNNDigits.m script)

Figure 15: Left : Output of the first layer for the CNNex.m script. Right : Handwritten digits to classify for the CNNDigits.m script

We can see that the network has learned to recognise some edges. If we look at the bottom left filter at Figure 15a, we can see that this filter is characterised by vertical lines of colours in order black - white - black - white. So, the output of this filter will be maximal when the input is similar to that pattern. If we look at the first five layers of the network architecture, we can observe that the **first layer** is just to take the images as input. The output of the first

[9] the CNN used is AlexNet

layers is (227,227,3). Then, the **second layer** is a convolutional layer with 96 filters of size [11 11] with stride [4 4] : we will build 96 feature maps of the input by convolving the image with a filter of size 11x11. We also use a stride of 4, so when sliding the filter accross the image, we will move it by 4 pixels. The padding used is "same', thus the output will be compressed to have as dimension : $dim_{out} = (dim_{in} - filtersize + 2 * padding)/stride \rightarrow [55, 55, 96]$. Then, the **third layer** is a ReLU activation, output is of dimension [55,55,96]. Then, the **fourth layer** is a cross channel normalization, output dimension is [55,55,96]. After that, the **fifth layer** is a max pooling layer : it allows to compress the data. With a 3x3 max pooling and a stride [2 2], the output dimension is $\frac{55-3}{2} + 1 = 27 \rightarrow$ [27,27,96]. We can confirm the dimensions we have found with the Matlab function *analyzeNetwork(convnet.Layers)*.

At the end of the netwotk, we have a fully connected layer of size 4096, and an output layer of size 1000 (corresponding to the 1000 classes). Therefore, we have 4096*1000 = 4 096 000 connections between the fully connected layer and the output layer. If we apply directly the input to the output layer, in a MLP framework for example, we would have 227*227*3*1000 = 154 587 000 connections ! On the other hand, using a CNN will result in much less connections. This is the advantage of CNN compared to MLP : it uses local connectivity, by building representation by looking at only a small area of the image at the time. In a convolutional layer, we said previously that we convolve a filter accross the image. Another way to interpret this is that have the same weights for the neurons of the same feature map. This way, the number of parameters needed is dramatically reduced : we have only one convolution kernel for all the neurons of the layers, and not a different kernel for every neuron.

3.3.2 CNNDigits.m

Here, trying several CNN architectures, we will classify a handwritten digits dataset. We will compare the performance of the different CNN architectures. The data is 28x28 black and white images, shown at Figure 15b. The network has an input of dimension [28 28 1], corresponding to the 28x28 images ("1" correspond to the number of channels, here we have only one channel because the images are black and white). The output of the network is a layer with 10 neurons, using a softmax activation. Between the input and the output layers, we have convolution and pooling layers. The activation function used is ReLU. We tried different configuration, presentation at Table 6.

ID	Architecture	Test accuracy [%]
1	Conv(5,12) ; maxpool(2,stride,2) ; Conv(5,24)	81.92
2	Conv(5,12) ; Conv(5,24)	96.80
3	Conv(5,64) ; maxpool(2,stride,2) ; Conv(5,32)	96.04
4	Conv(5,64) ; Conv(5,32)	98.20
5	Conv(5,128) ; maxpool(2,stride,2) ; Conv(5,64)	99.16
6	Conv(5,128) ; Conv(5,64)	98.96
7	Conv(5,128) ; maxpool(2,stride,2) ; Conv(5,64) ; maxpool(2) ; Conv(5,32)	96.80

Table 6: Accuracy of CNN for different configuration

The best performing architecture is Architecture 5, with a test accuracy of 99.16%. We can make several interesting observations :

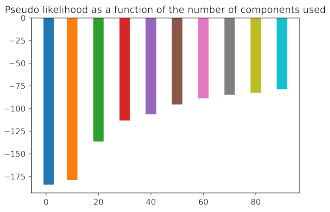
- For the low complexity networks, pooling layers does not help : we can see that the Architecture 2 has an accuracy of 96.80 compared to Architecture 1 which has an accuracy of 81.92, while Architecture 2 is the same network as Architecture 1 but without pooling layers. This can be interpreted as the network being already too simple, compressing even more the representation does not help.
- Pooling layers helps for high complexity networks : it allows to compress the representation, avoid overfitting and increase the speed of the training. Architecture 5 has a better performance than Architecture 6 for example.
- Increasing the complexity of the network helps, but after a while performance decreases : we can see that the Architecture 7 is the most complex, but not the best performing one. What happens is that the network start to overfit on the training set.

4 Generative models

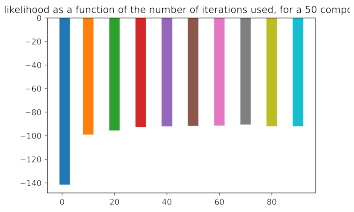
4.1 Restricted Boltzmann Machines

Restricted Boltzmann Machines can be viewed as a stochastic version of the Hopfield Networks, where the state of a neuron is equal to 1 with a certain probability. Learning a RBM is thus learning the parameters such that the probability distribution represented by the RBM fits the training data as well as possible. We have visible units (inputs and outputs) and hidden units. In a RBM, compared to a classical Boltzmann Machine, there is no hidden-to-hidden connections, which makes the training of the network simpler. One of the elegant feature of the RBM is that the energy function E we minimize, ignoring some offset terms, is in the form $-v^T W h$, with v being the input, W the interconnection matrix and h the hidden units. This term can be seen in two ways : if v is given, find h and W that maximally correlate with the input v (therefore train the network). On the other hand, if h is given, generate v that maximally correlate h and W (therefore generate new data).

The probability distribution represented by the RBM is defined in terms of the energy function E, as being $P(v, h) = \frac{1}{Z} e^{(-E)}$. From this probability distribution, we can obtain $P(v|h)$ and $P(h|v)$. $P(h|v)$ is used when the input layer is given, and we want to maximize the likelihood of the hidden layer, i.e. *train the network*. $P(v|h)$ is used when we know the hidden layer, and we want to maximize the likelihood of the input layer, i.e. *generate new data*. To simulate an RBM, we will use the BernoulliRBM function of sklearn[10]. As a metric, we use the pseudo-likelihood of the data, which is the likelihood of the data given the parameters of the RBM model. When training the model, we will select the parameters in order to maximize this likelihood. At Figure 16a and Figure 16b, results of experiments are shown. Figure 16a shows the likelihood of the data as a function of the number of component of the RBM, for a learning rate of 0.01 during 20 iterations. Figure 16b shows the likelihood of the data as a function of the number of iteration, for a RBM with 50 components trained with a learning rate of 0.01. From Figure 16a, we can observe that increasing the number of components increases the performance of the network. However the gain of performance is reduce after a network with 100 components, and we expect that for a even higher number of components the performance will not increase anymore. From Figure 16b, we can observe that the performance increases until the 30th iteration, and after it remains the same.



(a) Likelihood by nb of component



(b) Likelihood by nb of iteration

Figure 16: Likelihood by number of components and by number of iterations

In RBM training, Gibbs sampling is used. Gibbs sampling is a method to generate samples from a multi-variate distribution. The idea is to generate samples of each variable, given already sampled values of the other variables. The application of Gibbs sampling for RBM training is as follow : we start with random values for the visible units. Then, with each Gibbs step, we update the hidden units according to $p(h|v)$ and then update the visible units according to $p(v|h)$. To see the effect of the number of steps of Gibbs sampling we take, let's visualise reconstructed unseen test images for different values of Gibbs steps. At Figure 17, we compare reconstruction of unseen test images for a RBM with 5 component (left plots) and for a RBM with 50 component (right plots). From top to bottom, we use 1, 5, 10 and 20 Gibbs sample steps. First, we can observe that the 5 component RBM generate very noisy images, while the 50 component RBM generate quite accurate images. We can also observe that when training with more Gibbs steps, the images becomes more sharp.
Gibbs sampling is not computationally efficient : another technique called **Contrastive Divergence** is often preferred.

At Figure 18, we observe the reconstruction of missing pixels with RBM. At Figure 18c, we can observe a relatively accurate reconstruction using a RBM with 100 units after removing pixels from row 11 to 12, while at Figure 18b the

[10]This function assume a Bernouilli distribution for the visible and the hidden units. Note that this is a downfall of Boltzmann Machines : we have to assume a distribution while in some applications it is difficult to estimate the distribution beforehand

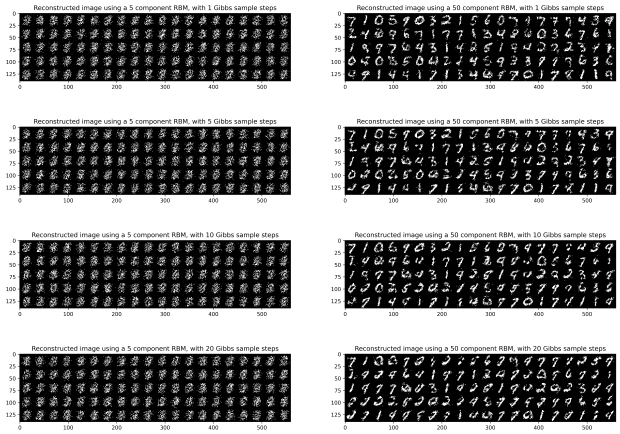


Figure 17: Reconstruction of unseen images for different Gibbs steps, with Left : RBM with 5 components and Right: RBM with 50 components (right)

reconstruction is very noisy when using an RBM with only 20 units : the number "6" is reconstructed as "0". Note that we remove only one row of pixels. At Figure 18d and 18e, we can observe the effect of removing more rows. At Figure 18d, we remove rows from positions 0 to 10 (from the top). We can observe that the RBM reconstructs the "6" as a "0", similarly to using a RBM with only 20 units. On the other hand, it is able to reconstruct the "0" and the "9" quite accurately. At Figure 18e, we remove rows from positions 20 to 30 (from the bottom). Again, the "6" is reconstructed as a "0", while the "9" is also reconstructed to a form similar to "0", given by the fact that when removing pixels from the bottom of a "9", the form becomes similar to "0". We can conclude that the RBM with more units is more accurate, removing more rows makes the reconstruction difficult and the position removed has an influence.

(a) Original image

(b) Reconstruction using a RBM with 20 units, removing pixels from row 11 to row 12

(c) Reconstruction using a RBM with 100 units, removing pixels from row 11 to row 12

(d) Reconstruction using a RBM with 100 unit, removing pixels from rows 0 to 10

(e) Reconstruction using a RBM with 100 unit, removing pixels from rows 20 to 30

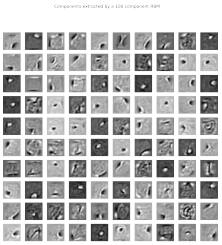
Figure 18: Reconstruction of an image representing the numbers "069"

4.2 Deep Boltzmann Machines

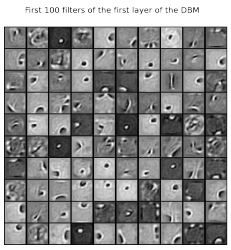
A Deep Boltzmann Machine is the extension of RBM with multiple layers. We use a pretrained DBM on the MNIST dataset to compare the performance with RBM. At Figure 19, we can see the components of each model after training. We can observe that the components of the RBM, at Figure 19a, are quite similar to those of the first layer of the DBM, at Figure 19b : both groups of components capture basic features. However, the second layer of DBM, shown at Figure 19c, seems to capture more complex features.
If we look at the reconstructed digits by the DBM at Figure 20 , we can see that even using a single Gibbs step, it is able to generate much more accurate digits compared to RBM

4.3 Generative Adversarial Networks

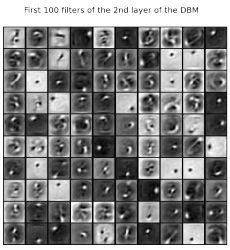
The Generative Adversarial Networks (GANs) is the idea to make compete two networks, namely a Generator that samples from a latent space p(z) to generate samples in the input space that approximate the actual distribution p(X) and a Discriminator that receives an input either from a generated sample from the Generator or a real sample and outputs the probability of the received sample being real or generated. The two networks compete in a zero-sum game manner, that is the Generator is trained to trick the Discriminator, while the Discriminator is



(a) Components of the RBM after training



(b) Components of the first layer of the DBM after training



(c) Components of the second layer of the DBM after training

Figure 19: Comparison of components of RBM and DBM

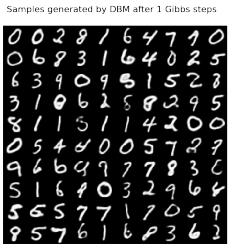
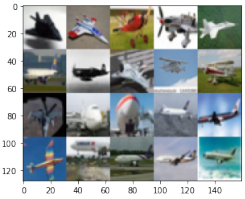
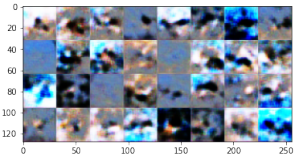


Figure 20: Reconstruction of unseen images with DBM

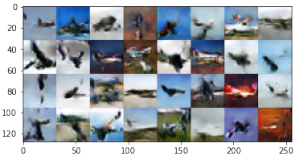
trained to avoid being tricked. DCGAN is the special case where we use convolutional layers to build the Generator and the Discriminator. We have trained a DCGAN on the CIFAR dataset to be able to generate *fake* samples from the class 0, namely the "aeroplane" class of the dataset, shown at Figure 21a.



(a) Real aeroplane images from the dataset



(b) Fake samples after 500 epochs



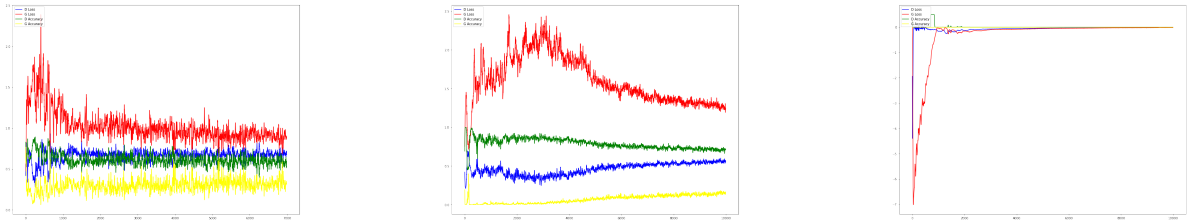
(c) Fake samples after 7000 epochs

Figure 21: Real aeroplane images from the dataset, and fake images generated with DCGAN

At Figure 21b and 21c, we can observe the samples generated by the DCGAN. Starting from noisy images as shown in Figure 21b, the generator becomes more and more accurate in order to increase its objective function which is to trick the discriminator. At Figure 21c, the images are realistic. At Figure 22a, we can observe the training curves of the DCGAN. The training is inherently unstable because we train two networks with opposed objective functions. To find a good local minima, we must have two networks with similar expressiveness. If the Generator is too much expressive compared to the Discriminator, the Discriminator will always be tricked, and the Generator will not be able to improve itself to be able to trick the Discriminator. Similar observations can be made in the case where the Discriminator is too much expressive compared to the Generator. To have a stable training, both models need to compete with each other in a *fair* manner, which is difficult to achieve.

4.4 Optimal transport

Optimal Transport is a theory to minimise the cost of transportation. Let's give an apocalyptic example : the city of Leuven is invaded by zombies. We have to protect the students ! Each faculty will produce anti-zombie vests. We have to deliver these vests to all the students who are all located in different bars (it is Thursday night). We known the amount of vests that can be produced by each Faculty, and the amount of students in each bar :



(a) Training of the DCGAN on CIFAR dataset

(b) Training of the minimax GAN on the digit dataset

(c) Training of the WGAN on the digit dataset

Figure 22: Training of the DCGAN, of the minimax GAN and of the WGAN

these can be represented by probability densities. Optimal Transport will be finding where each vest should go to minimize the total transport cost. The optimal transport is often called the Wasserstein metric. Another metric called Sinkhornn distance add a kind-of regularization term, where we can decide a trade-off between giving each bars equal amount of vests or delivering only closest bars in our example. In Machine Learning, we often use the optimal transport to find the statistical divergence between two probability distribution, i.e. how they differ. Here, instead of training a RBM with maximum likelihood as we did before, we will minimize the Wasserstein distance between the data distribution and the model distribution. At Figure 23, we use optimal transport to transfer the colours between two images. The first column correspond to the original image, the second column correspond to the use of Wasserstein Distance and the third column correspond to the use of the Sinkhorn distance. We observe that the result with the Sinkhorn distance is more realistic. We don't just swap pixels here : we find the optimal way of transferring one color distribution to another. The advantage is that compared to OT, just swapping the pixels does not take into acount the distribution of the colors, i.e. that the neighbouring pixels should be coherent etc.

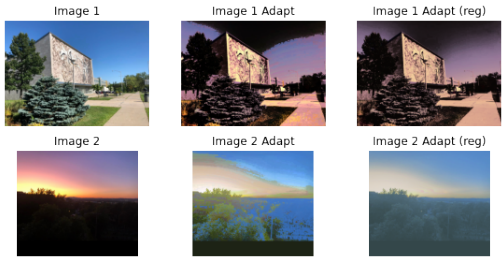


Figure 23: From left to right : original images, transfer with Wasserstein Distance, transfer with Sinkhorn distance

4.4.1 Minimax GAN vs Wasserstein GAN

WGAN, compared to GAN, has an objective function being to minimize the Wasserstein distance between the training data and the generated data. This improves the stability of the network. At Figure 24a, we can observe the results of training a GAN on the mnist dataset, while at Figure 24b we observe the results of training a WGAN on the same dataset. We can observe that the WGAN is able to produce less noisy images. At Figure 22b and 22c, we can observe the training curves of the GAN and the WGAN, and conclude that training a WGAN is much more stable.

(a) Minimax gan results after 10000 epochs

(b) Wgan results after 10000 epochs

Figure 24: Fake samples generated from the mnist dataset