Artificial Neural Networks And Deep Learning

Student Name : Somshree Mukherjee
Student Number : R0813317
Email : somshree.mukherjee@kuleuven.be

August 2020

Contents

1 Exercise Session 1 : Supervised learning and generalization

1.1 Backpropagation in feedforward multi-layer networks

A feedforward multi-layered neural network is considered one of the simplest and most general methods for supervised learning. The operation of this kind of network can be divided into two steps, the feedforward step and then the backpropagation step. In the feedforward part, an input pattern is applied to the input layer and its effect propagates layer by layer through the network until an output is reached. After this, the actual output of the network obtained, is compared to the expected output and an error signal is computed for each of the output nodes. Since each intermediary hidden layer has contributed to the error signal in the output nodes, this error signal is propagated backwards to the nodes in the hidden layer that contributed to the error. This process is repeated per layer until every node in the network has an associated error signal corresponding to its relative contribution in the overall error. After this the nodes can use the corresponding error values to update the value of the weights until the network arrives at a steady state where the training patterns can be encoded. The process of backpropagation tries to find the minimum value of the error function by using various training algorithms and the weights that finally minimise the error function is considered to be a solution to the supervised learning problem.

1.1.1 Function approximation: comparison of various algorithms

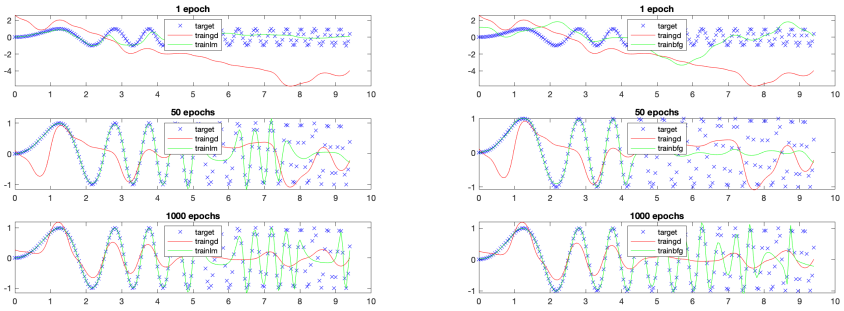
Function approximation is a technique using which we can estimate an unknown function, by using some previously available data. In a dataset that comprises of some inputs and outputs, supervised learning algorithm learns how to map inputs to outputs based on what it learns from the mapping of some sample inputs and outputs. This kind of "mapping function" is what is approximated by a supervised learning algorithm. In neural networks the error between expected output and predicted output is minimised during the "training" process.

In the first exercise, we take a function

$$y = sin(x^2) \quad \text{for} \quad x = 0 : 0.05 : 3\pi$$

We try to approximate it using a neural network with one hidden layer using various different algorithms. In particular, we have used three algorithms - Gradient Descent, Levenberg-Marquardt and BFGS quasi-Newton. We then make a compare of the performance of the Gradient Descent algorithm with respect to Levenberg-Marquardt and BFGS quasi-Newton algorithms for 1, 50 and 1000 epochs.

From the Fig 1.1.1.1 (left) we can see that Levenberg-Marquardt fits the function much better than Gradient Descent in all three cases with different epochs. The fitting also somewhat improves for Levenberg-Marquardt with the



Gradient Descent vs Levenberg-Marquardt Gradient Descent vs BFGS quasi-Newton

Fig 1.1.1.1 Comparison for 1,50 and 100 epochs

increase in epoch numbers, however Gradient Descent still performs poorly. From the second plot (Fig 1.1.1.2 (right)) as well we see that Gradient Descent performs poorly in comparison to BFGS quasi-Newton and its performance does not improve even with the increase in epochs.

Comparing the three algorithms together (Fig 1.1.1.2) for epochs 1, 50 and 1000, we can see that Levenberg-Marquardt performs better than both Gradient Descent as well as BFGS quasi-Newton at all epoch numbers.

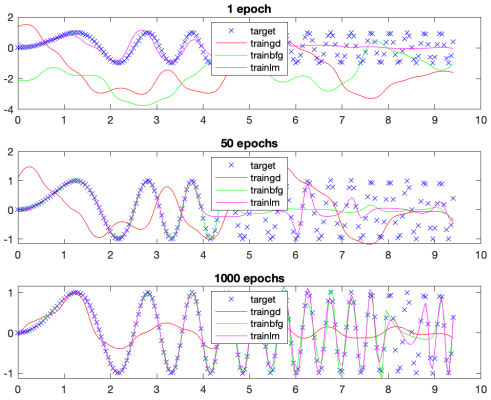
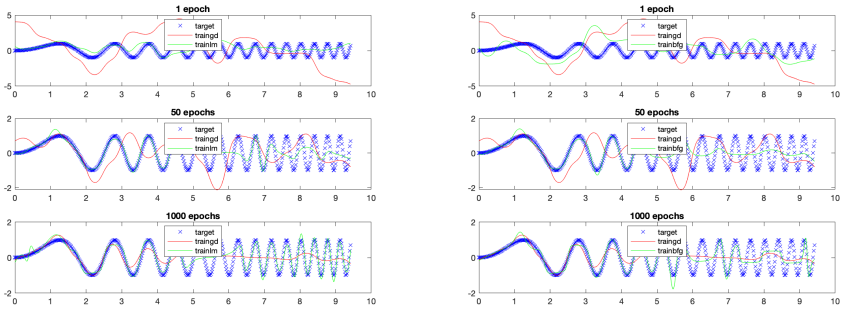


Fig 1.1.1.2 : Gradient Descent vs BFGS quasi-Newton vs Levenberg-Marquardt (1, 50 & 1000 epochs)

1.1.2 Learning from noisy data: generalization

In the second part of the exercise, we add a noise to the data with amplitude 0.5.



Gradient Descent vs Levenberg-Marquardt Gradient Descent vs BFGS quasi-Newton

Fig 1.1.2.1 Comparison for 1,50 and 100 epochs - Noisy data

We make a similar comparison as the previous section, where we compare the fitting performance of the Gradient Descent algorithm with respect to Levenberg-Marquardt and BFGS quasi-Newton algorithms for 1, 50 and 1000 epochs.

In this case as well we make similar observations. Fig 1.1.2.1 show that Gradient Descent does not perform well in comparison to Levenberg-Marquardt (left) and BFGS quasi-Newton algorithms(right) at each of the epoch levels. In each algorithm, the fitting seems to improve with the increase in epoch numbers and among the three algorithms, Levenberg-Marquardt again performs the best (Fig 1.1.2.2). It is interesting to note that in general, the fitting was better for the three algorithms for the noisy data than in the first section, and hence the approximation was better in this case.

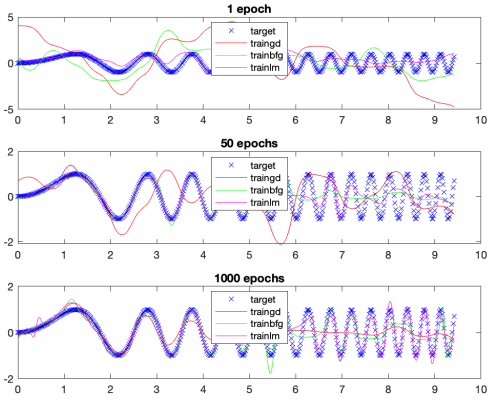


Fig 1.1.2.2 : Gradient Descent vs BFGS quasi-Newton vs Levenberg-Marquardt (1, 50 & 1000 epochs) : Noisy Data

1.1.3 Personal regression example

In this exercise, the objective was to approximate a nonlinear function using a feedforward artificial neural network. The nonlinear function is unknown but we have an already given set of 13600 datapoints uniformly sampled from it.

We have to first construct the dataset using the digits of or student number. The student number used in this case is R0813317. The largest five digits in descending order used are

$$d_1 = 8, d_2 = 7, d_3 = 3, d_4 = 3, d_5 = 1$$

According to the given formula,

$$T_{new} = (d_1T1 + d_2T2 + d_3T3 + d_4T4 + d_5T5)/(d_1 + d_2 + d_3 + d_4 + d_5)$$

We calculate

$$T_{new} = (8T1 + 7T2 + 3T3 + 3T4 + 1T5)/(8 + 7 + 3 + 3 + 1)$$

where

$$T_1, T_2, T_3, T_4, T_5$$

are from the input data file and is representative of five independent non-linear functions.

We divided our dataset into three parts one for each, training set, validation set and test set, in order to avoid correlation between the samples and we also make sure that the training set has a random distribution of data points from throughout the dataset. This can be seen in Fig 1.1.3.1.

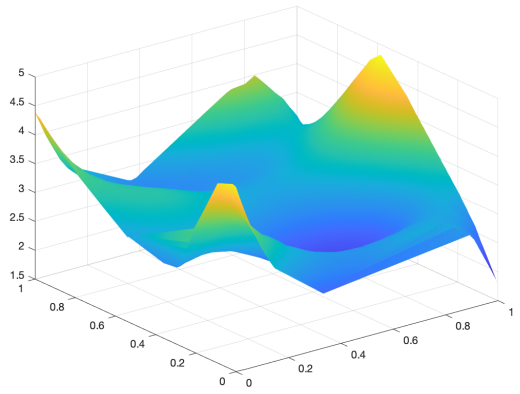
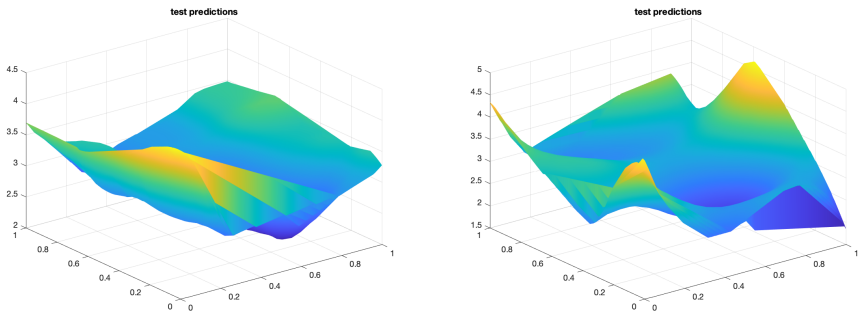


Fig 1.1.3.1 : Surface plot for original training data points

We then use a neural network with one hidden layer and 20 neurons, for 500 epochs. We use the transfer function `tansig` since it is a very popularly



Gradient Descent Levenberg-Marquardt

Fig 1.1.3.2 : Surface plot for predicted training data points

used transfer function in regression problems. We also compare the performance of this neural network for two algorithms - Gradient Descent and Levenberg-Marquardt.

In the case of Gradient Descent we observe that the predicted test data (Fig 1.1.3.2 (left)) is very different from the original test data (Fig 1.1.3.1). Whereas, in the case of Levenberg-Marquardt (Fig 1.1.3.2 (right)), the original and predicted data points are quite similar, and hence we conclude that Levenberg-Marquardt performs better than Gradient Descent. We also calculate and compare the Mean Squared Error(MSE) for our training, validation and test data set for these two algorithms (Table below) and we observe that the MSE for Levenberg-Marquardt is considerably lower in all cases than Gradient Descent, and hence we can conclude again that Levenberg-Marquardt is the better algorithm for our training problem.

Mean Squared Error		
	Levenberg-Marquardt	Gradient Descent
Training	2.1784e-06	0.1007
Validation	4.7909e-06	0.1079
Test	4.6516e-06	0.1148B8

1.2 Bayesian Inferences

1.2.1 Using trainbr to compare previous datasets

In this exercise we use the training algorithm `trainbr` which is based on Bayesian Regularization backpropagation. It works on a Levenberg-Marquardt optimization technique which further minimises the mean squared error and produces a network that generalises well.

We repeat the exercises for section 1 using this algorithm, and we make a comparison of Bayesian Regularization with Levenberg-Marquardt. We observe

from Fig 1.2.1.1 and Fig 1.2.1.2 Bayesian Regularization outperforms Levenberg-Marquardt in both cases of data without noise and with noise, and we can hence conclude that Bayesian Regularization is superior of the two.

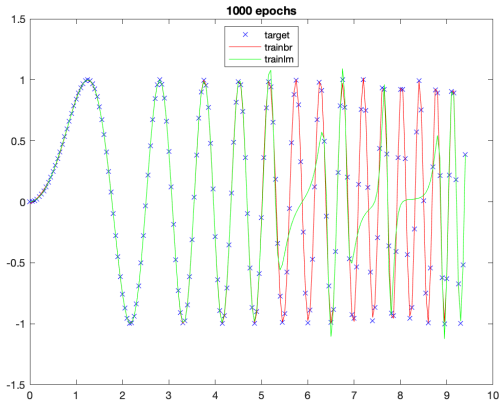


Fig 1.2.1.1 : Levenberg-Marquardt vs Bayesian Regularization - No Noise

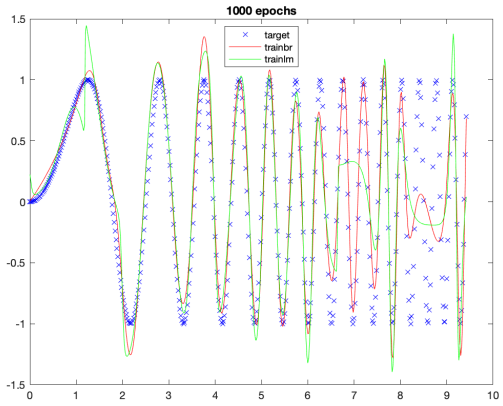


Fig 1.2.1.2 : Levenberg-Marquardt vs Bayesian Regularization - With Noise

2 Exercise Session 2 : Recurrent neural networks

2.1 Hopfield Network

A Hopfield network is a specific type of single-layered recurrent neural network and involves simulating human memory through pattern recognition and storage. Unlike a regular feedforward neural network, the nodes in a Hopfield Network are both inputs and outputs and fully interconnected. Since the network can store patterns and retrieve them based on a partial input because of the interconnections, they are useful in applications involving noisy or partial data.

In the first part of this exercise, we create a two-neuron Hopfield network with attractors

$$A = [+1 + 1; -1 - 1; +1 - 1]^T$$

We then define 10 random input points and simulate the Hopfield network for these points in 50 iteration steps. From the plot in Fig 2.1.1 we can see that all possible states are contained within the boundaries and most of the input points converge to the three attractors. However we also notice that some of the input points converge to a fourth attractor state, which leads us to believe that it is possible for the real number of attractors to be bigger than the number of attractors used to create the network. Usually all the input points are able to reach the attractors in about 20 iterations.

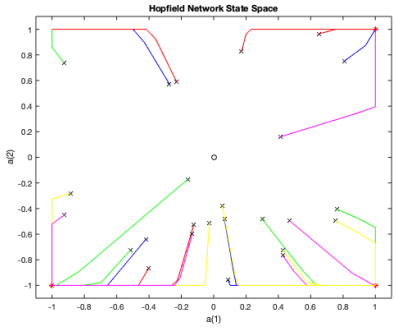


Fig 2.1.1 : Two neuron Hopfield Network with random input

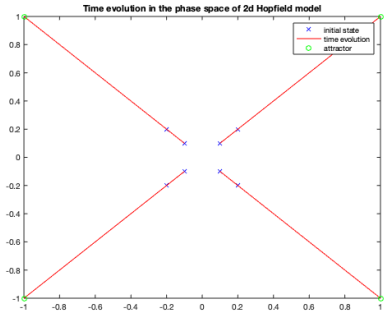


Fig 2.1.2 : Two neuron Hopfield Network with symmetric input

In the second part, we generate the input points with a predefined vector of points with high symmetry while using the same attractors as before.

$$InputPoints = [+0.2 + 0.2; +0.2 - 0.2; -0.2 + 0.2; -0.2 - 0.2; +0.1 + 0.1; +0.1 - 0.1; -0.1 + 0.1; -0.1 - 0.1]^T$$

From Fig 2.1.2 we observe that we get four attractors out of which three are the same as the ones stored in the network at creation and one of the attractors (-1,1) is added by the network.

In the third part, we implement a Hopfield network for three neurons with similar experiments as the previous sections. We use attractors

$$A = [+1 + 1 + 1; -1 - 1 + 1; +1 - 1 - 1]^T$$

and generate 10 random input points and simulate our Hopfield network for 50 timesteps. From Fig 2.1.3 we observe again, that all possible states are contained within the boundaries and the points converge to the three attractors.

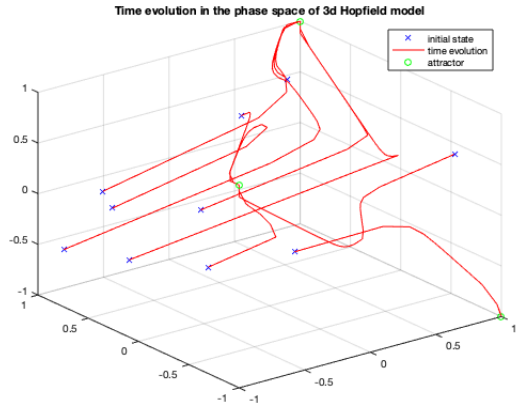


Fig 2.1.3 : Three neuron Hopfield Network with random input

In the fourth part, we have a function `hopdigit` that creates a Hopfield network which has as attractors the handwritten digits 0,...,9. We test the ability of the network to correctly retrieve these patterns by giving some noisy digits as input to the network.

The function `hopdigit` is of the form `hopdigit(noise,numiter)` where `noise` represents the level of noise that will corrupt the digits and is a number between 0 and 10 and `numiter` is the number of iterations that the Hopfield network will run for the noisy digits.

For `noise = 2` and `numiter = 10`, we observe from Fig. 2.1.4 that the Hopfield network is able to recognise the digits quite well. For `noise = 5` and `numiter = 10`, we observe from Fig 2.1.4 that the network is no longer able to correctly recognise some of the digits. So keeping the noise level same we increase the `numiter = 50`, and we observe in Fig 2.1.4 that the network is still not able to correctly recognise some of the digits.

This shows us a limitation of the Hopfield Network, where it is not always able to reconstruct a pattern. This is explained by the fact that every trained set of data can be imagined as a local minima in a surface. So by increasing the amount of datapoints into the network, spurious states of local minima might be introduced, meaning that if two local minima are too close, they might "fall" into each other to create a single local minima which does not correspond to any

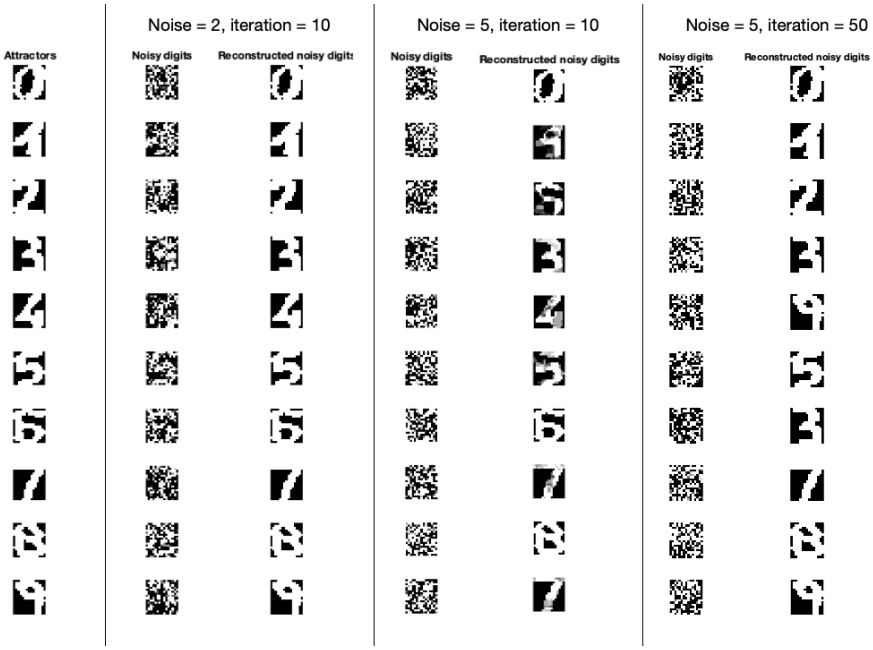


Fig 2.1.4 : Hopfield Digit Recognition for varying noise levels and iterations

training sample, while forgetting about the samples it is supposed to memorize, which can cause confusion/error in the network. Hence with the introduction of noise in the sample of the digits, the network was not able to identify the digits correctly.

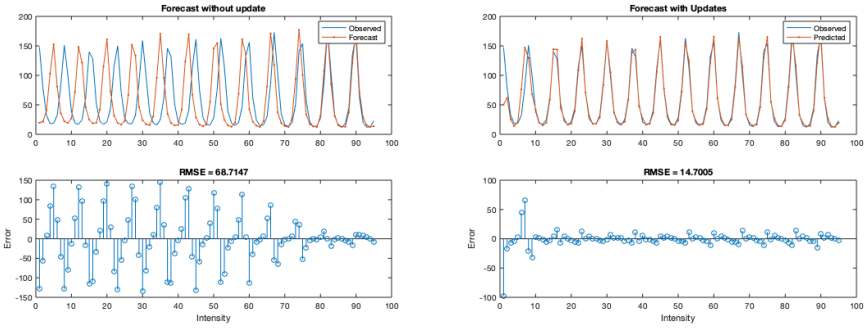
2.2 Long Short-Term Memory Networks

LSTMs are a special kind of RNN capable of learning long term dependencies. These networks can "remember" information for long periods of time and hence are useful for making time series predictions. In this exercise we try to model the Santa Fe laser intensity dataset. We train an LSTM model by trying various combinations of hidden layers, epochs and lag values and arrive at the final set of hyperparameters to use for the LSTM.

In Fig 2.2.1, we can see the effect of changing various hyperparameters. In particular we observed that increasing the lag value resulted in the RMSE and loss value to decrease much more smoothly than with a lag value of 1. Reducing the number of hidden layers ro 100 for the same lag value, resulted in a lesser elapsed time and there was still an overall decrease in RMSE and Loss error but the decrease was not as smooth as in the previous case. Increasing the number of epochs to 400 increased the total elapsed time considerably while the decrease in RMSE and loss error was not much better than in the case of 250 epochs for the same lag value and hidden layers. Hence, for finally training our LSTM,

we keep the following optimum hyperparameters : Lag = 5, Number of hidden layers = 200 and Number of Epochs = 250

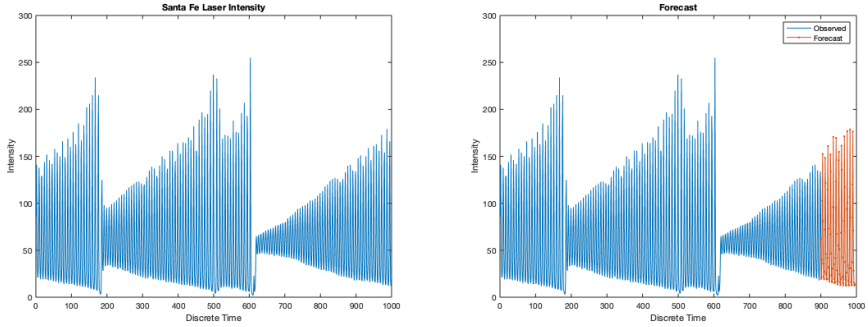
We then try to predict the test set using this trained LSTM. We compare the results of prediction with update state with the results of prediction without update state and observe that the prediction with update state had a considerably lower RMSE value (Fig 2.2.2). A comparison of the original Santa Fe dataset and a time-series prediction of it can be seen in Fig 2.2.3. When comparing LSTMs to RNN, LSTM gives much better results for this problem set since essentially LSTM is a better version of RNN that allows it to pick information and look at it from a larger collection of information. The higher accuracy however can often come at the cost of high complexity and operation costs.



Without update state
RMSE = 68.7147

With update state
RMSE = 14.7005

Fig 2.2.2 : Time series prediction for Santa Fe dataset



Original dataset

Time series Forcast

Fig 2.2.3 : Time Series Forcast for the Santa Fe dataset

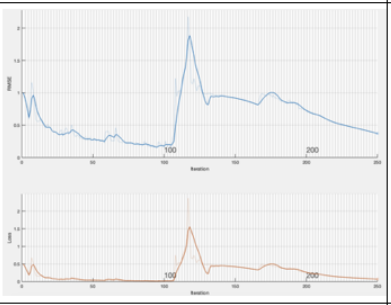
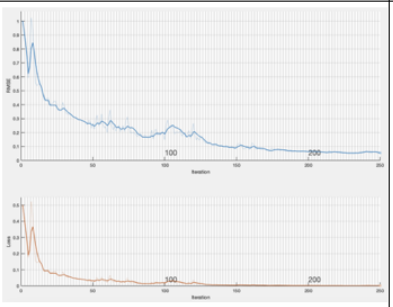
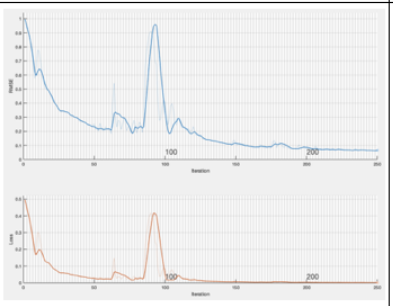
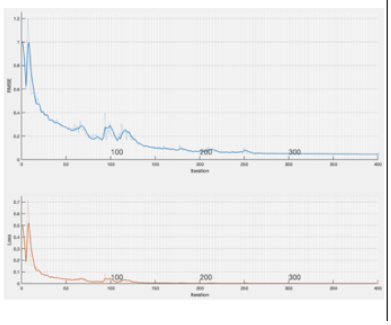
RMSE LOSS	Lag	Hidden Layers	Epoch	Elapsed Time
	1	200	250	2 mins 50 s
	5	200	250	2 mins 46 s
	5	100	250	1 mins 33 s
	5	200	400	4 mins 53 s

Fig 2.2.1 : Changing various hyperparameters to train LSTRM model

3 Exercise Session 3 : Deep feature learning

3.1 Principal Component Analysis

Principal Component Analysis is common used to reduce the dimensionality of large data sets. Reduction of dimensionality can come at a cost of accuracy, and hence with PCA, we try to reduce the number of variables of the data set while still preserving as much information as possible. In this method we first standardize our dataset so that each of the variables can contribute equally to the analysis. This is done so that the variances in the initial variables is bot too high. else the variables with larger range will dominate the ones with a smaller range and can lead to biased results. We then compute the covariance matrix for our dataset. With this step we aim to understand the relationship between the variables of the input by seeing how they are varying from the mean with respect to each other. Then we compute the eigen vectors and eigen values for the covariance matrix in order to identify the principal components. Principal components are the new variables computed from the dataset such that they contain a mixture of the initial variables. This mixture is done in such a way that most of the information from the intial variables is compressed into the components in descending order of eigen values. Hence we manage to reduce dimensionality without losing much of the information from our initial variables and we can then discard some of the last few components as they would have the least information. In this way we create a feature vector consisting of the components with the maximum information and finally reconstruct our data in terms of the principal components.

3.1.1 Redundancy and Random Data

In the first part of this exercise we compare the reduction of random data to the reduction of highly correlated data and we notice that for highly correlated data, most of the information is obtained in just the first few eigen values and hence it becomes easy to reduce the dimensions of such a dataset. However for random data, where the data does not have much correlation, each eigen value retains a large amount of information and hence reducing the dimension in this case becomes practically impossible. Hence we conclude that PCA is not an effective method for data sets with low correlation between their components.

3.1.2 Principal Component Analysis on Handwritten Digits

In the second part of this exercise, we apply Principal component analysis on handwritten images of the digit '3' taken from the US Postal Service database. In the first step we calculate the mean and plot it for each column of the dataset as shown in Fig 3.1.2.1. From the figure, we can see that the mean varies across the columns and does not show any evidently recognizable patterns. In the next step we compute the covariance matrix for our dataset and plot the eigenvalues as in Fig 3.1.2.2.

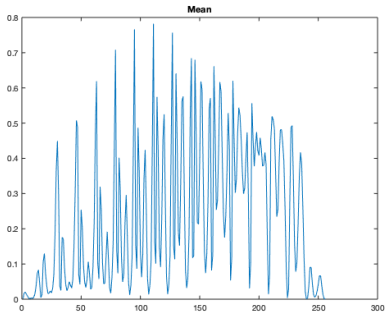


Fig 3.1.2.1 :
Mean for the original dataset

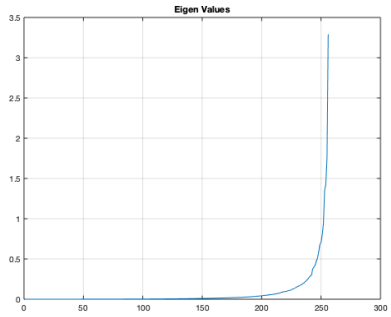


Fig 3.1.2.2 :
Eigen values from the Covariance matrix

We then compress the dataset by projecting it onto one, two, three and four principal components and reconstruct the images from these compressions. The reconstructed images along with the original image are shown in Fig 3.1.2.3

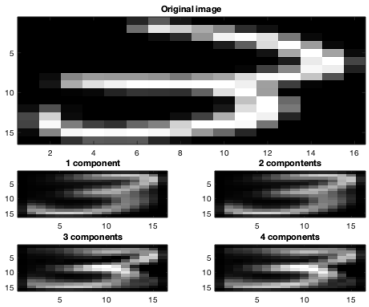


Fig 3.1.2.3 : Original image vs Reconstructed Images

In the next step we compress the entire dataset by projecting it onto 1..50 principal components and then reconstruct the images. We then measure the reconstruction error as a function of the number of principal components as shown in Fig 3.1.2.4. From this plot we can see that the reconstruction error is reduced considerably as the number of principle components to project to is increased. Ideally if we use 256 principal components which is basically using all the pixels in the image, then our reconstruction error should be zero. However, in our case we observed that the reconstruction error for 256 principal components was 6.0621e-16, which maybe because during the calculation of the covariance matrix some numbers might have been truncated or some decimal values might have lost during the float calculations.

In the last step we calculate cumulative eigen values for the 256 components as shown in Fig 3.1.2.5 and we observe that with the increase in components beyond approximately 100, the amount of information remains relatively the

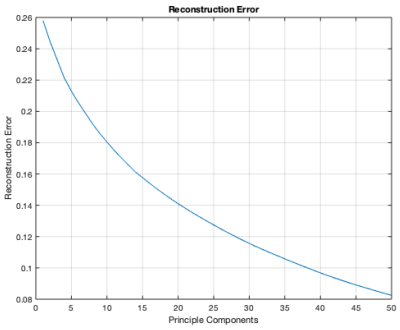


Fig 3.1.2.4 :
Reconstruction Error vs
number of components

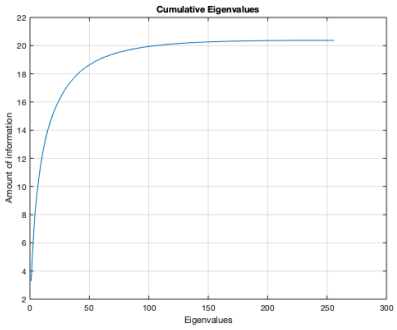


Fig 3.1.2.5 :
Cumulative Eigen values vs
Amount of Information

same and hence we can say that it is enough to use about 100 components for our analysis.

3.2 Stacked Autoencoders

Autoencoders are a technique of unsupervised learning where we use neural networks for a task of representation learning. In this technique, we try to compress data into a bottle-necked network (encoder) and then let the network learn how to reconstruct data back from this compressed version (decoder), into a representation which is as similar to the original input as possible. In stacked autoencoders, we can have multiple stacks of layers consisting of encoders and decoders where outputs of each layer is connected to the inputs of the successive layer.

In the first part of this exercise we try to classify digits using stacked autoencoders. The stacked auroencoder has three parts in which we first train the first autoencoder layer, then the second autoencoder layer and then finally a softmax layer for supervised learning. In each autoencoder layer we set the size of the hidden layer smaller than the input size. Finally we perform a fine-tuning of our network using backpropagation to improve the results.

In Fig 3.2.1 we can see the input image of the digits. After training the first autoencoder for 784 input images using 100 neurons we can see the result in Fig 3.2.2 where it plots the weight associated with each feature. We can can see that the autoencoder learnt to represent features such as curls and stroke patterns from the digit images.

Next we train the second autoencoder in a similar way as the first one. For the second autoencoder, we use the features that were generated from the first autoencoder as the training data for the second. We also decrease the number of neurons to 50 so that the second autoencoder can learn an even smaller representation of the input. In this way we reduce dimensionality of our dataset. We then train a final softmax layer in a supervised fashion to classify

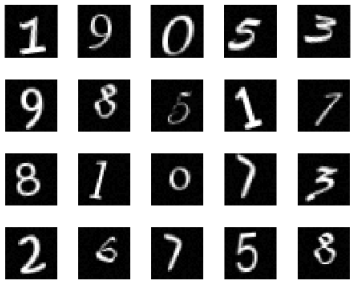


Fig 3.2.1 : Input image of the digits

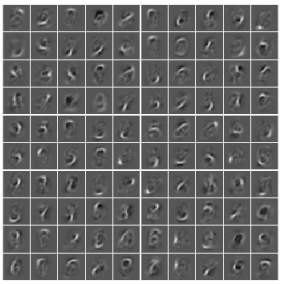


Fig 3.2.2 : Weight plot for features after first autoencoder layer

these 50-dimensional vectors into different digit classes. These three neural networks stacked together as shown in Fig 3.2.3 form a deep neural network that we have trained to classify digits.

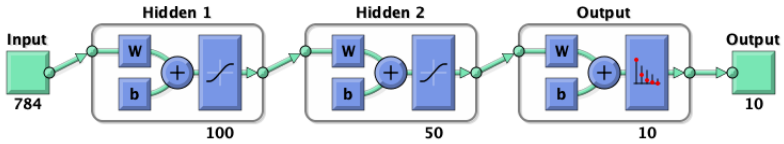


Fig 3.2.3 : Deep Neural network consisting of stacked autoencoders and a softmax layer

At the end of these series of training processes we observe that the accuracy of classification is 82.84%. In order to improve our accuracy, we then do a fine-tuning of our neural network by performing a backpropagation on the whole multilayer network. By doing this we observe that the accuracy of classification has improved considerably to 99.64%. When we compare this performance to a normal neural network with 1 hidden layer, the accuracy of classification in this case is 95.84% which is lower than what we achieved with the fine-tuned stacked autoencoder. Even if we use 2 hidden layers in a normal neural network the accuracy of classification is 96.58, which is still lower than what was achieved with the fine-tuned stacked autoencoder. Even after changing (reducing) the various parameters like number of epochs or number of neurons in the first hidden layer of the autoencoder, it was observed that the accuracy of classification was still better than that achieved through normal neural networks with 1 or 2 hidden layers. If we add one more layer to the deep neural network itself, we see a decrease in the accuracy instead of an increase, which is probably due to overfitting caused by the extra complexity. If we perform fine-tuning on this deep neural network with an extra layer, our accuracy improves but is still lower than what we previous achieved with combination of 2 autoencoders and 1 softmax layer, which leads us to believe that for our particular problem set,

having 2 layers in the deep neural network is optimum.

3.3 Convolutional Neural Networks

A Convolutional Neural Network is a type of deep learning algorithm that can take some input image and through assignment of some weights and biases to various aspects of the image or objects in the image, it is able to differentiate one from the other. This method has a higher level of accuracy as compared to MLPs even for complex images. Through the application of relevant filters, a CNN is able to successfully understand the spatial and temporal dependencies of the image.

In this exercise, we implement a Convolutional Neural Network as a category classifier and use it to identify images from the Caltech101 dataset. But instead of operating on all the images of this dataset, we only use three categories - airplanes, ferry, and laptop and the image classifier identifies amongst these three categories.

We use a pre-trained CNN called AlexNet that has been trained on the ImageNet dataset which has 1000 object categories and 1.3 million training images. We inspect the network architecture for this CNN and we observe (Fig 3.3.1) that it was 23 layers, with the input images in the first layer and a series of layers consisting of Convolution, ReLU, Cross Channel Normalization and Max Pooling repeated several times followed by some Fully Connected Layers and finally a penultimate layer consisting of the Softmax function and finally the Classification Output Layer.

```
23x1 Layer array with layers:

   1   'input'                'Image Input'                   227x227x3 images with 'zerocenter' normalization
   2   'conv1'                'Convolution'                   96 11x11x3 convolutions with stride [4  4] and padding [0  0  0  0]
   3   'relu1'                'ReLU'                          ReLU
   4   'norm1'                'Cross Channel Normalization'   cross channel normalization with 5 channels per element
   5   'pool1'                'Max Pooling'                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
   6   'conv2'                'Convolution'                   256 5x5x48 convolutions with stride [1  1] and padding [2  2  2  2]
   7   'relu2'                'ReLU'                          ReLU
   8   'norm2'                'Cross Channel Normalization'   cross channel normalization with 5 channels per element
   9   'pool2'                'Max Pooling'                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
  10   'conv3'                'Convolution'                   384 3x3x256 convolutions with stride [1  1] and padding [1  1  1  1]
  11   'relu3'                'ReLU'                          ReLU
  12   'conv4'                'Convolution'                   384 3x3x192 convolutions with stride [1  1] and padding [1  1  1  1]
  13   'relu4'                'ReLU'                          ReLU
  14   'conv5'                'Convolution'                   256 3x3x192 convolutions with stride [1  1] and padding [1  1  1  1]
  15   'relu5'                'ReLU'                          ReLU
  16   'pool5'                'Max Pooling'                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
  17   'fc6'                  'Fully Connected'               4096 fully connected layer
  18   'relu6'                'ReLU'                          ReLU
  19   'fc7'                  'Fully Connected'               4096 fully connected layer
  20   'relu7'                'ReLU'                          ReLU
  21   'fc8'                  'Fully Connected'               1000 fully connected layer
  22   'prob'                 'Softmax'                       softmax
  23   'classificationLayer'  'Classification Output'         crossentropyex with 'n01440764' and 999 other classes
```

Fig 3.3.1 : 23 layers of the Convolutional Neural Network

In layer 1 we have the input images with dimension `227x227x3`. In layer 2 we use a convolutional filter of size `11x11x3` and we use `96` of these, to identify 96 features as shown in Fig 3.3.2.

To calculate the size of the output we use this formula :

$$O = \frac{W - K + 2P}{S} + 1$$

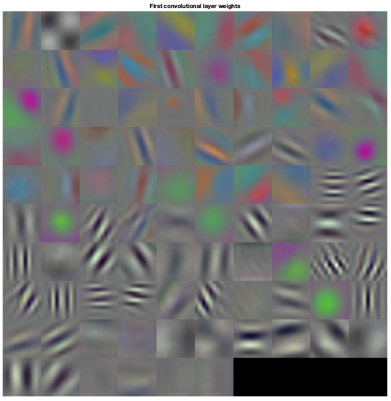


Fig 3.3.2 : 96 weighted features identified after the second layer

where O is the output size (height or length), W is the size(height or length) of the input, K is the size of the filter, P is the size of the padding and S is the stride.

According to this, the size of the output at the end of the second layer is `(227 - 11 + 2*0)/4 + 1 = 55`. Hence we have a `55x55x96` sized output. The third layer is a ReLU layer that implements a Rectified linear activation function, which is a piece-wise linear function that will output the input directly if it is positive and output zero otherwise. The fourth layer is a cross-channel normalization that normalises the output received from the ReLU layer. Both these layers hence, do not affect the dimensions of the input, so the input size to the fifth layer will be the same as that obtained at the end of layer two, which is `55x55x96`. In layer 5, a max pooling function is implemented, that down-samples the feature maps by summarising the features in blobs of the feature map, which makes the final feature map even more robust. To calculate the size of the output the Max Pooling layer we use the following formula :

$$O = \frac{(W - F)}{S} + 1$$

where O is the output size (height or length), W is the size(height or length) of the input, F is the spatial extent parameter, and S is the stride.

Using this we calculate the size of the output at the end of the second layer as `(55 - 3)/2 + 1 = 27`. Hence the size of the output after Layer 5 will be `27 x 27 x 96`, which will be the size of the input to layer 6.

We can also see from Fig 3.3.2 that the number of neurons used in the last layer of the classification task is 1000 and this is because this CNN was trained to identify 1000 classes. In our case since we only identify 3 classes, it is sufficient to use three neurons, one for each of the classes that we wish to identify (airplanes, ferry and laptop).

4 Exercise Session 4 : Generative models

4.1 Restricted Boltzmann Machines

Boltzmann Machines are non-deterministic generative deep learning models that have hidden nodes and visible nodes and do not have an output node. In these kind of networks all the input nodes and hidden nodes are interconnected with each other unlike the other networks we learnt in previous exercises. It is because of this reason the nodes in this kind of networks are capable of sharing information among themselves and hence self-generate sufficient data to learn on. Because of this, Boltzmann machines are also called Deep Generative Models and fall into the category of unsupervised deep learning. A Restricted Boltzmann machine is a two layered neural network and has generative capabilities, which means that they are able to learn a probability distribution over its set of inputs. It is "restricted" in the sense that the two layers in an RBM correspond to a visible layer and a hidden layer and the nodes in the visible layer are connected to every node in the hidden layer, but no two nodes in the same group are connected to each other. This restriction makes RBMs more efficient than general Boltzmann machines.

In the first part of the exercise we implement an RBM for the MNIST dataset. For this we train the RBM on the dataset for different parameters - number of components which is the number of binary hidden units and number of iterations which is the number of times the training data set will be trained upon. For `number of components = 10` and `number of iteration = 10`, we observe (Fig 4.1.1) that the time taken for training increases with increasing number of iterations which implies that the results will be more accurate and hence the reconstruction error will be less. We also see that the pseudo-likelihood increases with the increase in number of iterations which implies that better trained RBM will have a higher accuracy of reconstruction. However for the same number of iterations, increasing the number of components increases the time taken for training considerably and the pseudo-likelihood is also higher than in the previous case which implies that increasing the number of components trains the RBM better and hence increases the accuracy of reconstruction (Fig 4.1.2). Increasing the number of iterations overall also increased the overall accuracy of reconstruction of the RBM.

This can be seen in the reconstruction of the image in the Gibbs Sampling step. For reference the original test image is shown in Fig 4.1.3. For an RBM trained with 10 components, and Gibbs steps = 10, the reconstruction is extremely lossy (Fig 4.1.4), but on using an RBM that was trained for 100 components we see that even with 10 Gibbs steps, the reconstruction is fairly good (Fig 4.1.5). Increasing the number of Gibbs steps however seem to have a detrimental effect on the quality of reconstruction (Fig 4.1.6).

In the next part of this exercise, we try to identify the role of the RBM hyperparameters on the performance in terms of accuracy of reconstruction images with missing parts. The number of hidden units captures features from the training set, so the more number of hidden units, then higher will be the

```
rbm = BernoulliRBM(n_components=10, learning_rate=0.01, random_state=0, n_iter=10, verbose=True)
rbm.fit(X_train)
```

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -196.74, time = 2.49s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -196.54, time = 2.88s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -196.54, time = 2.87s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -196.41, time = 2.90s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -196.40, time = 2.89s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -188.95, time = 3.00s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -187.56, time = 2.96s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -186.88, time = 3.00s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -186.83, time = 3.01s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -187.51, time = 2.95s
BernoulliRBM(batch_size=10, learning_rate=0.01, n_components=10, n_iter=10,
             random_state=0, verbose=True)
```

Fig 4.1.1 : RBM Training for 10 components and 10 iterations

```
rbm = BernoulliRBM(n_components=100, learning_rate=0.01, random_state=0, n_iter=10, verbose=True)
rbm.fit(X_train)
```

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -112.04, time = 10.97s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -98.47, time = 12.36s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -91.87, time = 12.31s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -89.06, time = 12.43s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -85.56, time = 12.31s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -84.18, time = 12.37s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -81.96, time = 12.40s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -81.36, time = 12.44s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -81.01, time = 12.33s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -80.16, time = 12.30s
BernoulliRBM(batch_size=10, learning_rate=0.01, n_components=100, n_iter=10,
             random_state=0, verbose=True)
```

Fig 4.1.2 : RBM Training for 100 components and 10 iterations

number of captured features and hence the reconstruction of unseen images will be more accurate. The learning rate in an RBM affects how much the network can adapt the error correction per iteration. It helps determine the step size towards a local optimum. Hence a higher learning rate will make the network learn faster and maybe cross this optimum value and a smaller learning rate will slow down the learning which can be inefficient (Fig 4.1.7). By increasing the number of iterations, we allow the network to train for longer and hence increase accuracy.

When we remove `n` number of rows from the images and use our RBM trained with 100 components and 0.05 learning rate and try to reconstruct the original image, we observe that for small number of removed rows, the RBM is able to reconstruct the images fairly well. But when a large number of rows are removed the reconstruction is not so accurate (Fig. 4.1.8).

Fig 4.1.3 : Reference : Original test image

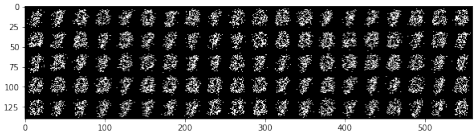


Fig 4.1.4 : Image Reconstruction with 10 components and 10 Gibbs steps

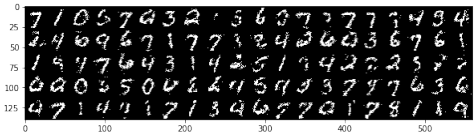


Fig 4.1.5 : Image Reconstruction with 100 components and 10 Gibbs steps

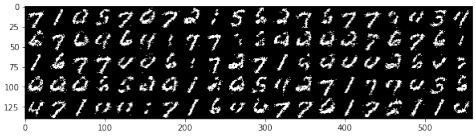
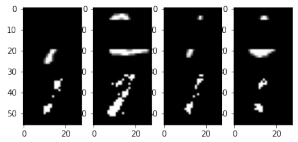
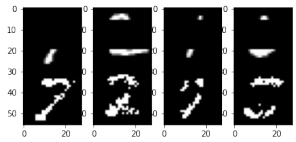


Fig 4.1.6 : Image Reconstruction with 100 components and 50 Gibbs steps

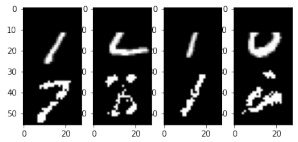


100 components, 10 Gibbs steps,
0.01 learning rate

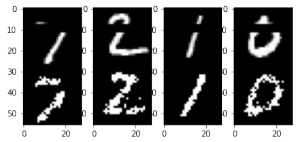


100 components, 10 Gibbs steps,
0.05 learning rate

Fig 4.1.7 : Reconstruction of unseen image for different learning rate



Removed rows 0-12



Removed rows 8-12

Fig 4.1.8 : Reconstruction of images with missing rows

4.2 Deep Boltzmann Machines

A deep Boltzmann machine is like a series of Boltzmann machines that are stacked on top of each other. In this exercise we compare the performance of the DBM trained on the MNIST dataset with the previously used trained RBM for the same dataset. We can see the filter weights for RBM (Fig 4.2.1) is much lesser than the filter weights for both layers of DBM (Fig. 4.2.2). Hence we can see that DBM being a kind of stacked RBM, is able able to extract features much better. Among the two layers of the DBM, we can see that the first layer extracts initial features and hence has a lower feature weight (Fig 4.2.2 (left)) while the second layer extracts more details and hence has a higher filter weight (Fig 4.2.2 (right)).

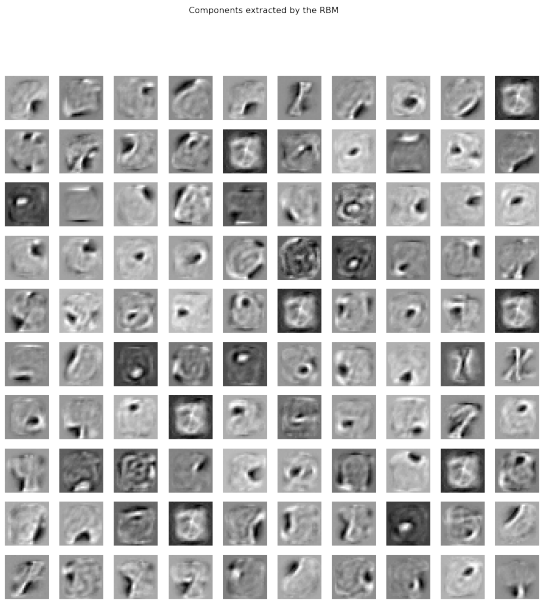
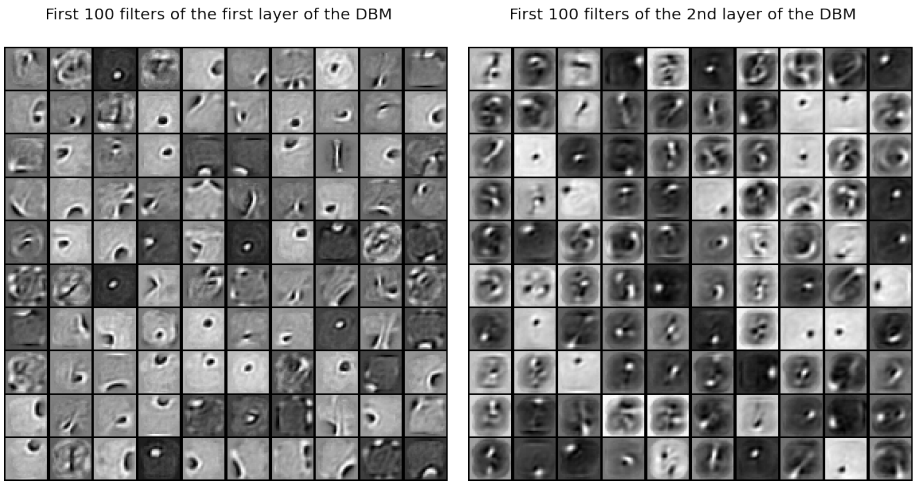


Fig 4.2.1 : Filter weights for RBM trained on MNIST training data

When we sample images from DBM and compare it to the images sampled by RBM for the same number of Gibbs steps = 100, we observe that DBM has a much better result as compared to RBM (Fig 4.2.3). This is because RBMs have a much simpler inference process than DBMs, where the hidden units are grouped into a hierarchy of layers and there is a full connectivity between subsequent layers but no connectivity within the layers or between non-neighbouring layers. Hence in DBM the inferencing is done as an approximation and depends on multiple passes through the hidden layers. We need to have some kind of iterative sampling, which in this case is a Gibbs sampling, in order to know the state of the layers above and below.



Filter weights for Layer 1 of DBM Filter weights for Layer 2 of DBM

Fig 4.2.2 : Comparison of filter weights for 2 layers of DBM trained on MNIST training data

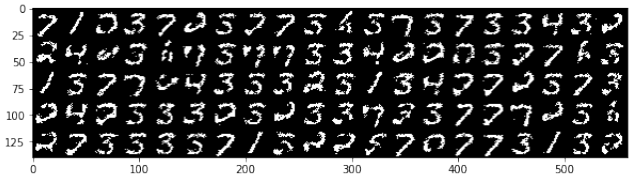


Fig 4.2.3 : Image quality for RBM (MNIST dataset)

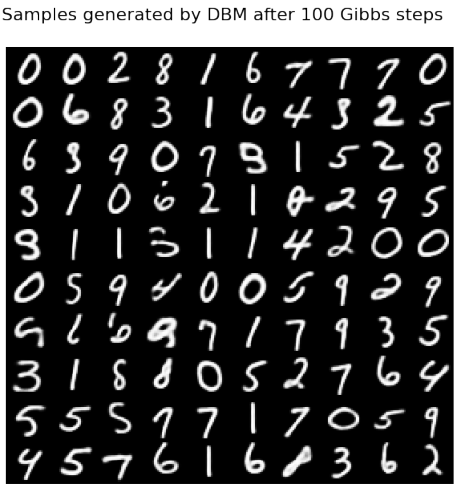
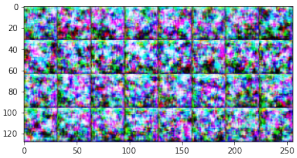


Fig 4.2.4 : Image quality for DBM (MNIST dataset)

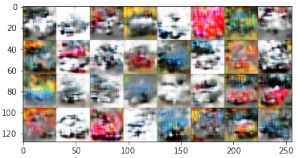
4.3 Generative Adversarial Networks

Generative Adversarial Networks or GANs are used in unsupervised learning techniques which implements two neural network - a generator and a discriminator that compete against each other in a zero-sum game (hence "adversarial"). The generator takes in some random numbers and returns an image. This generated image is then fed to a discriminator along with the actual ground-truth dataset. The discriminator then takes in both the real and fake images and returns probabilities - a number between 0 and 1 where o represents a fake and 1 represents a prediction of authenticity.

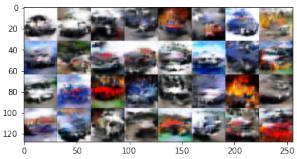
In this exercise we train a Deep convolutional generative adversarial network(DCGAN) on the CIFAR dataset which contains 10 classes with 6000 32 x 32 colour images per class. However we only use one class of this dataset for our exercise. After we have trained our DCGAN, we try to identify the Discriminative Loss, which is a measure of how well the discriminator was able to distinguish between real and fake images, and the Generative Loss, which is a measure of how well the generator was able to trick the discriminator into accepting a fake image as a real image.



Batch 51
D˙loss = 0.5145 , G˙loss = 0.9614



Batch 1551
D˙loss = 0.5945 , G˙loss = 0.8678



Batch 5001
D˙loss = 0.6825 , G˙loss = 0.994

Batch 10501
D˙loss = 0.6764 , G˙loss = 0.7952

Fig 4.3.1 : Training of GAN progressively over different batches

From the images in Fig 4.3.1, we can see that the instances of fake image generation from the generator become stronger with each back propagation and the generator is easily capable of tricking the discriminator. With respect to the generative loss and discriminative loss, the values keep varying over the batches which is how we expect it to be in this tugging game of the generator against the discriminator. However in general the generative loss value is higher in more cases than the value for discriminative loss which leads us to believe that overall the generator is usually able to trick the discriminator successfully.

4.4 Optimal transport

The Optimal Transport problem compares two probability distributions and identifies all possible ways to morph, transport or reshape the first distribution into the second distribution and associates a global cost every such transport using a local cost of transporting each of the components of the distribution, so as to find the least costly and most efficient computation for transport.

In this exercise we use two methods of optimal transport to transfer colours between two images. The scatter plot for colour distribution of the two images is shown in Fig 4.4.1. First we solve this problem by identifying the optimum distance as a Wasserstein metric which is a distance between two probability distributions (also called earth mover distance). It can be interpreted as how much volume needs to be transported in order to change one distribution into the other. In the second way, we identify the optimum distance as a Sinkhorn distance which is a modified form of the Wasserstein distance but additionally is a function of information entropy. This information entropy can be increased by making the distribution more homogeneous. The Sinkhorn distance works better than the Wasserstein distance for Optimal transport problems since there is no cost factor involved in the distribution and hence the distribution becomes homogeneous. The comparison of the image transformations for both Wasserstein distance and Sinkhorn distance are shown in Fig 4.4.2

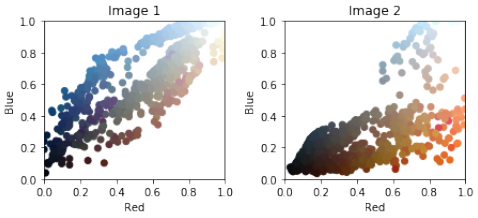


Fig 4.4.1 : Scatter plot for colours in both images

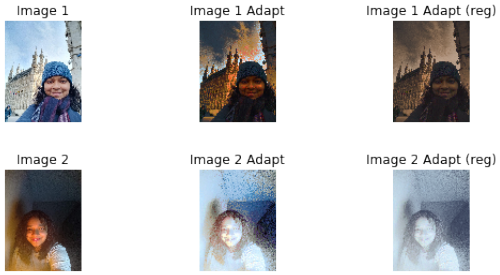


Fig 4.4.2 : Original Images, EMD Transport Images (Adapt) and Sinkhorn Tranport Images (Adapt Reg)

4.5 Wasserstein GAN

Wasserstein GAN is an extension of the Generative Adversarial Network that improves the stability when training the model and also provides a loss function correlated to the quality of the generated images. As an extension of GAN, WGAN tries to train the generator model to make a better approximation of the data distribution for a given training dataset. Unlike GAN which uses the discriminator to predict the probability of generated images being fake or real, WGAN replaces the discriminator with a "critic" that scores the realness or fakeness of a given image. This is because WGAN seeks to train the generator in a way that tries to minimize the distance between the distribution of the training dataset and the distribution in the generated samples. This makes WGAN more stable and less sensitive to choice of hyperparameters, and also relates the loss of the discriminator to the quality of the images generated by the generator. Wasserstein distance as a metric between probability distributions (also called the earth mover distance) can hence, be used for the Optimal Transport problem.

In this exercise we train a fully connected minimax GAN and a Wasserstein GAN on the MNIST dataset. Fig 4.5.1 compares the performance of the two GANs over different iterations. We can clearly see that there is a marked improvement in the images reconstructed by the WGAN in comparison to the ones generated by GAN. Even though training the GAN takes considerably more time, but the reconstruction of the images and stability is much better compared to GAN.

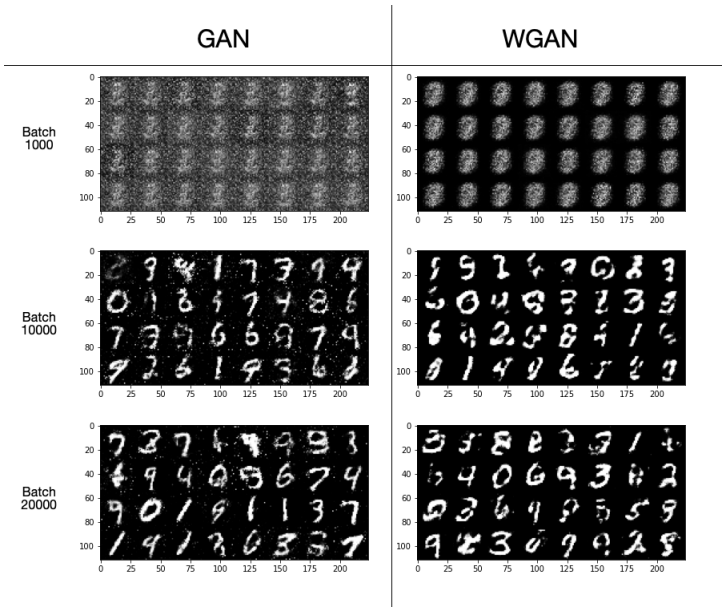


Fig 4.5.1 : Comparison of Image Reconstruction using GAN vs WGAN