KU LEUVEN

Master of Artificial Intelligence

Artificial Neural Networks Course Report

Name : Keerthitheja Samudrala Chandrasekar
Student Number : r0773368

Prof. Johan Suykens
Artificial intelligence
KU Leuven

Contents

1 Supervised learning & generalization **1**
1.1 Backpropagation in feedforward multi-layer networks 1
1.2 Personal regression 2
1.3 Bayesian Inference 4

2 Recurrent neural networks **6**
2.1 Hopfield Network 6
2.2 Long Short-Term Memory Networks 8
2.2.1 Time-series Prediction and Long short-term memory network 8

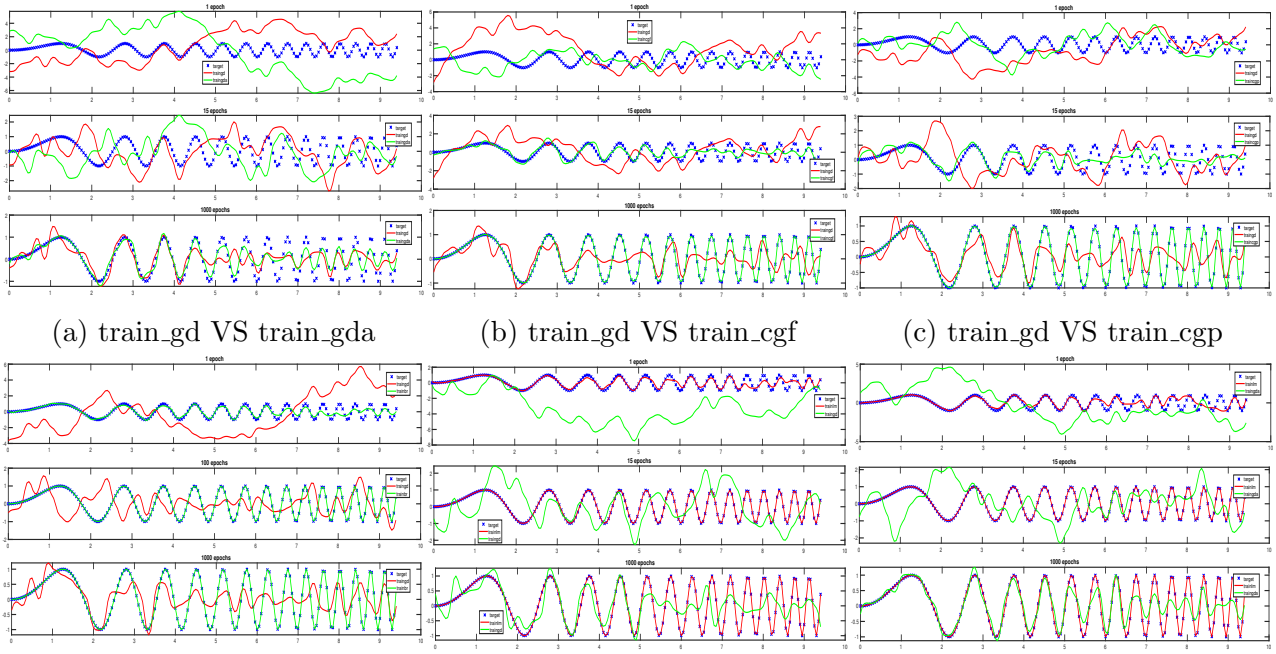
3 Deep feature learning **10**
3.1 Principal Component Analysis 10
3.2 Stacked Autoencoders 11
3.3 Convolutional Neural Networks 12

4 Generative models **15**
4.1 Restricted Boltzmann Machines 15
4.2 Deep Boltzmann Machines 17
4.3 Generative Adversarial Networks 17
4.4 Optimal transport 18

1. Supervised learning & generalization

1.1 Backpropagation in feedforward multi-layer networks

In this exercise we train a feed forward neural network to fit the function $y = sin(x^2)$ with and without noise. The universal approximation theorem states that under mild activation assumptions a feed forward neural network with a single hidden layer and finite number of neurons can approximate a continuous function well. Therefore, we train a network with single hidden layer and evaluate the performance using various training algorithms like gradient descent *(traingd)*, gradient descent with adaptive learning rate *(traingda)*, Fletcher-Reeves conjugate gradient *(traincgf)*, Polak-Ribiere conjugate gradient *(traincgp)*, BFGS quasi Newton *(trainbfg)*, Levenberg-Marquardt *(trainlm)* and Levenberg-Marquardt with Bayesian regularization *(trainbr)*. We compare the results of these algorithms with gradient descent. The function approximations of these algorithms trained for 1000 epochs is shown in the below figure 1.1.

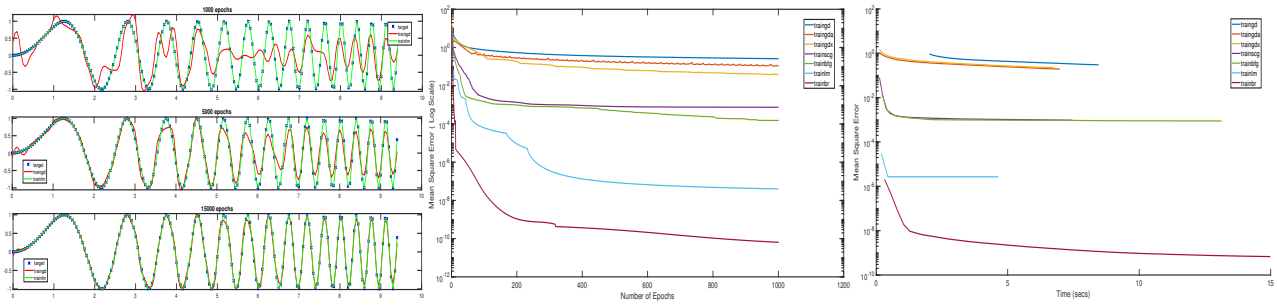


(a) train_gd VS train_gda (b) train_gd VS train_cgf (c) train_gd VS train_cgp

(d) train_gd VS train_br (e) train_gd VS train_lm (f) train_gda VS train_lm

Figure 1.1: Comparison of various training algorithm with gradient descent on input data without noise for [1;100;1000] epochs

It can be seen from the figure that gradient descent performs the least amongst the other training algorithms. Gradient descent with adaptive learning rate performs better than the normal gradient descent. The function approximations is done by reducing the mean square errors over the training epoch. If the number of epochs is increased then gradient descent also gives better approximations just like other algorithms as shown in the figure 1.2a. From figures 1.2b and 1.2c, it can be seen that training algorithm with Levenberg-Marquardt with



(a) train_lm VS train_gd for epochs [1000;5000;15000]

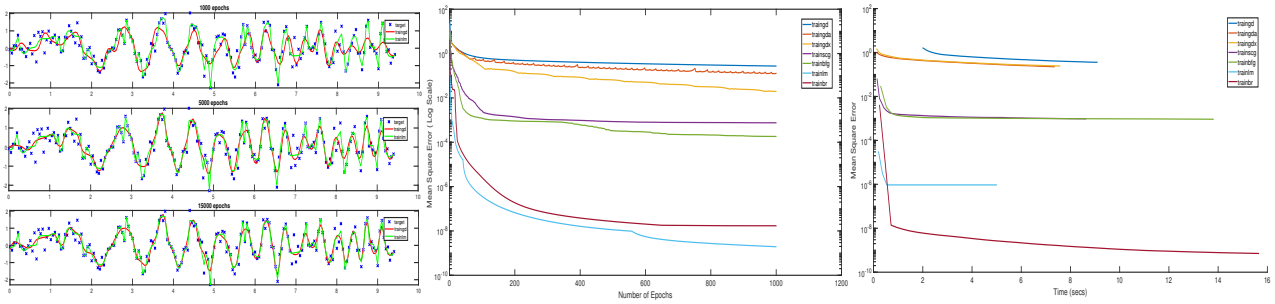
(b) MSE VS Number of training epochs

(c) MSE VS Training time

Figure 1.2: Performance of various training algorithms on input data without noise

Bayesian regularization *(trainbr)* has the least MSE but training time is slightly more. Overall Levenberg-Marquardt *(trainlm)* is the best with minimum training time and mean square errors.

Similar observations can be seen for the input data with noise added via $y = sin(x^2) + \sigma * randn(1, length(x))$, where σ is noise level and chosen as 0.5. the performance of Levenberg-Marquardt is best on this noisy data. Also, gradient descent gives good approximation if the number of epochs is increased as shown in figure. This is because for larger iterations, the convergence rate increases.



(a) train_lm VS train_gd for epochs [1000;5000;15000]

(b) MSE VS Number of training epochs

(c) MSE VS Training time

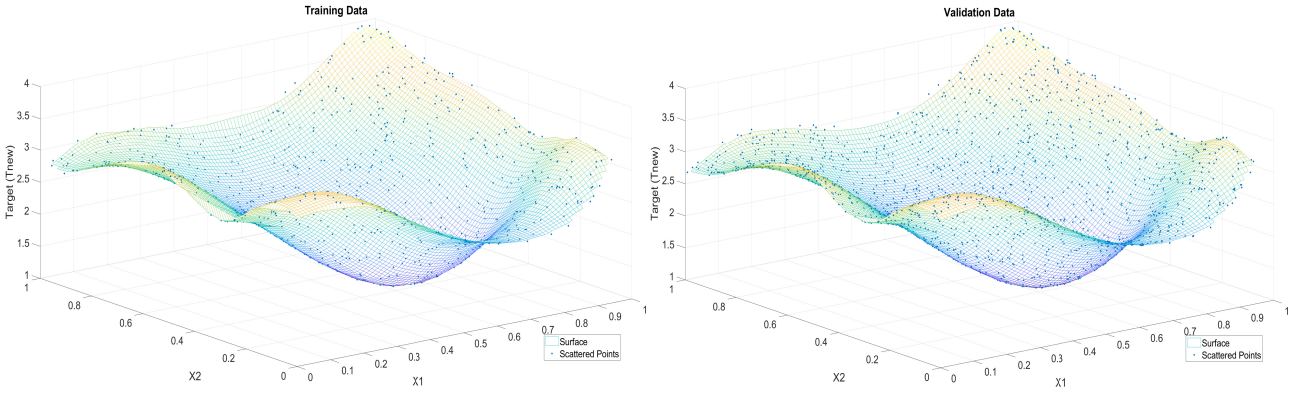
Figure 1.3: Performance of various training algorithms on input data with noise

From figure 1.3b it can be seen that both Levenberg-Marquardt with Bayesian regularization *(trainbr)* and Levenberg-Marquardt *(trainlm)* gives smaller mean square errors but Levenberg-Marquardt is faster as it takes lesser time for convergence of the mean the error over 1000 epoch as shown in figure 1.3c.

1.2 Personal regression

In this exercise we try to approximate an unknown non-linear function on a dataset with 136000 points. We only consider the first 3000 data points to create our train, validation and test sets. We randomly permutate the data points and chose 1000 points amongst the dataset for each of train, validation and test sets.The target data points is generated as $T_{new} =$

$(8T1 + 7T2 + 7T3 + 6T4 + 3T5)/(8 + 7 + 7 + 6 + 3)$. The train and validation scatter plots on a mesh grid is shown in the figure 1.4. For training the model we use both training and validation data sets. We train the model with [20 50 100] neuron combinations in the hidden layers using various training algorithms and transfer functions. Out of these models, we chose the model with best performance and least mean square error to predict the approximations on the test dataset. Some of the results of the training methods is given in the table



(a) Training data points (b) Validation data points

Figure 1.4: Scatter and Mesh plots of Training and validation data points

Training Algorithm	Network transfer function	Hidden units	MSE on train set	MSE on val set	Training Time
traingd	logsig	20	0.065107	0.063997	4.3129
traingd	tansig	20	0.040031	0.040949	4.2408
trainlm	logsig	50	8.645e-07	4.8764e-06	36.538
trainlm	tansig	50	1.4024e-06	4.6243e-06	6.6497
trainbr	logsig	50	2.233e-09	1.2061e-08	42.345
trainbr	tansig	50	1.0037e-09	1.3104e-08	41.626
trainlm	tansig	100	5.5275e-09	1.4636e-07	144.04
trainbr	tansig	100	2.805e-10	2.2196e-09	146.42

Table 1.1: Performance of a training algorithms in non-linear function approximation of the given data points for various configurations

From the table 1.1 it can be seen that both *trainlm* and *trainbr* algorithms gives smaller MSE's on train and validations. The method *trainlm* with 50 hidden units and *'tansig'* network transfer function takes smaller training time. The MSE of this configuration is less and acceptable even though *trainbr* has the minimum value. Therefore, we test our test set with this configuration for the non-linear function approximation. The predictions on the test set is shown in the figure 1.5. It can be noticed that the model performs at its best. To further improve the performance, we can tune the hyper parameters and use more hidden layers and hidden units with drop out nodes (to avoid over fitting).

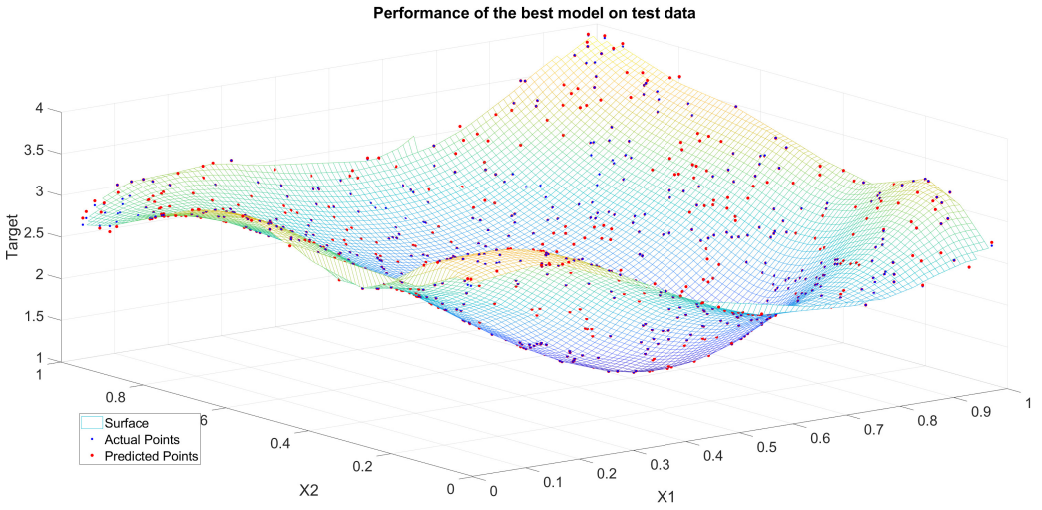


Figure 1.5: Non linear function approximates on test set

1.3 Bayesian Inference

(a) Performance on data without noise

Training Algorithm	Hidden units	MSE on training set	Training Time
'traingd'	30	0.41024	1.4715
'traincgf'	30	0.11332	2.9307
'trainlm'	30	0.049307	3.0955
'trainbr'	30	0.00016628	6.111
'traingd'	50	0.22564	2.1072
'traincgf'	50	0.021669	3.8808
'trainlm'	50	1.9356e-09	6.6776
'trainbr'	50	5.8591e-11	9.4141
'traingd'	100	0.28868	2.5079
'traincgf'	100	0.000343	4.3097
'trainlm'	100	2.9575e-16	0.94997
'trainbr'	100	5.8841e-11	5.4633
'traingd'	150	0.28617	2.4044
'traincgf'	150	2.3958e-05	2.9279
'trainlm'	150	9.2161e-17	0.21211
'trainbr'	150	4.9838e-06	39.717

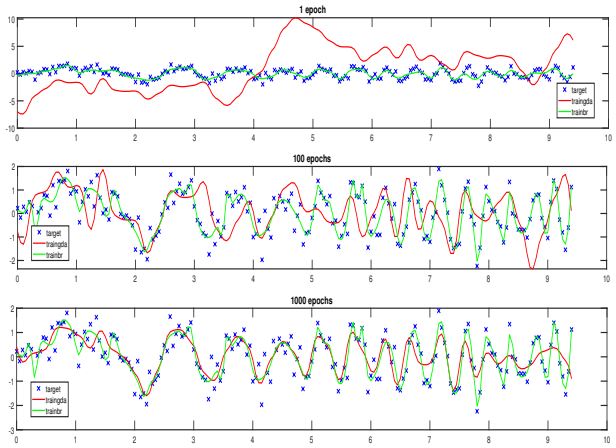
(b) Performance on noisy data with noise level $\sigma = 0.5$ (standard deviation)

Training Algorithm	Hidden units	MSE on training set	Training Time
'traingd'	30	0.45688	1.2805
'traincgf'	30	0.25429	2.3445
'trainlm'	30	0.12776	2.4669
'trainbr'	30	0.1832	3.1289
'traingd'	50	0.32496	1.1241
'traincgf'	50	0.14655	2.6011
'trainlm'	50	0.067438	3.8249
'trainbr'	50	0.12235	4.9785
'traingd'	100	0	0.31386
'traincgf'	100	0.028183	3.3844
'trainlm'	100	0.00016868	6.6878
'trainbr'	100	0.0036395	9.5772
'traingd'	150	0	0.34608
'traincgf'	150	0.0064246	4.1436
'trainlm'	150	2.5763e-27	10.262
'trainbr'	150	0.00016898	24.427

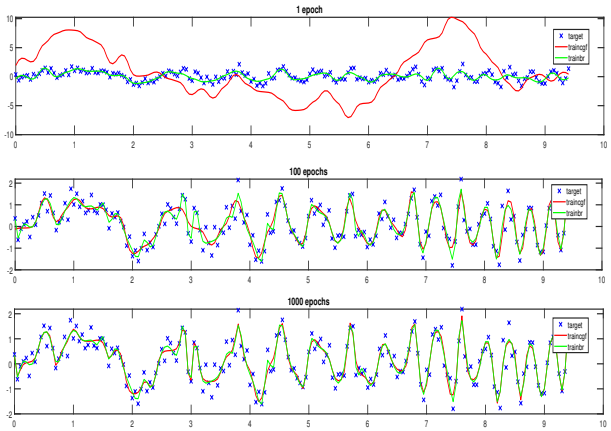
Table 1.2: Performance of training algorithms for input data with and without noise

In this exercise we evaluate the training algorithm *trainbr* i.e., Levenberg-Marquardt with Bayesian regularization similar to section 1 of this exercise. As seen in the earlier section,

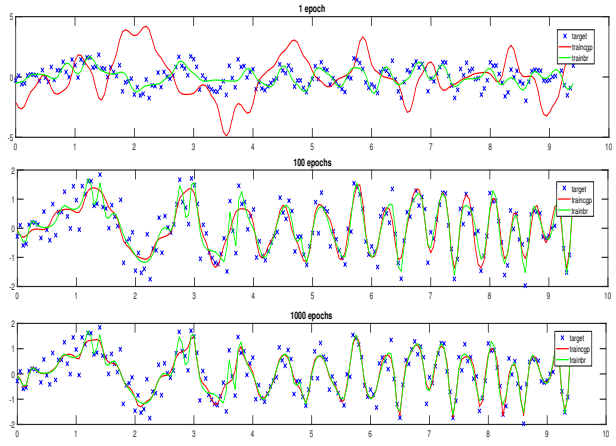
gradient descent does a poor job compared to other algorithms. Here, we compare the algorithm *trainbr* with others for [30;50;100;150] hidden units in its hidden layer. The results of the models are shown in table 1.2 and figure 1.6 respectively. From figure 1.6 it can be noticed that *train_br* approximates better than *train_gda*, *train_cgf* and *train_cgp* for input data with noise while *train_lm* approximates better than *train_br*. It was also noticed that for lesser number of hidden units the *train_br* model doesn't give good approximates. It works well in the case of input data without noise and gives a very small MSE whereas for input data with noise, *train_lm* outperforms all the models both in MSE and training time (results can be seen from table 1.2).



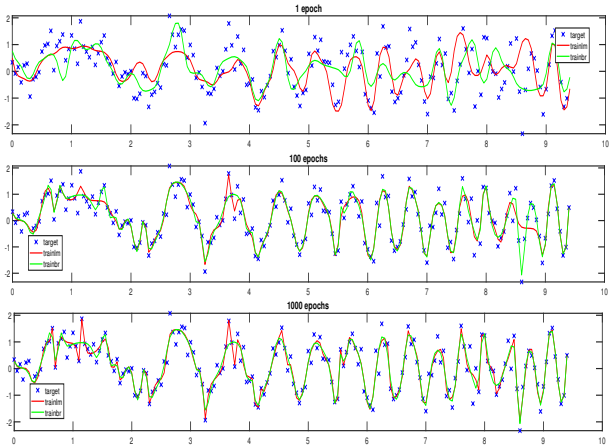
(a) train_br VS train_gda



(b) train_br VS train_cgf



(c) train_br VS train_cgp



(d) train_br VS train_lm

Figure 1.6: Comparison of various training algorithm with *trainbr* on input data for [1;100;1000] epochs

Over-parameterized networks take more time to train and the training with more units can only lead to over-fitting. Also, it can be noticed from table 1.2 that the overall performance is better for hidden units 50 by taking the training time, MSE, etc. into consideration than with higher number of units like 100 or 150.

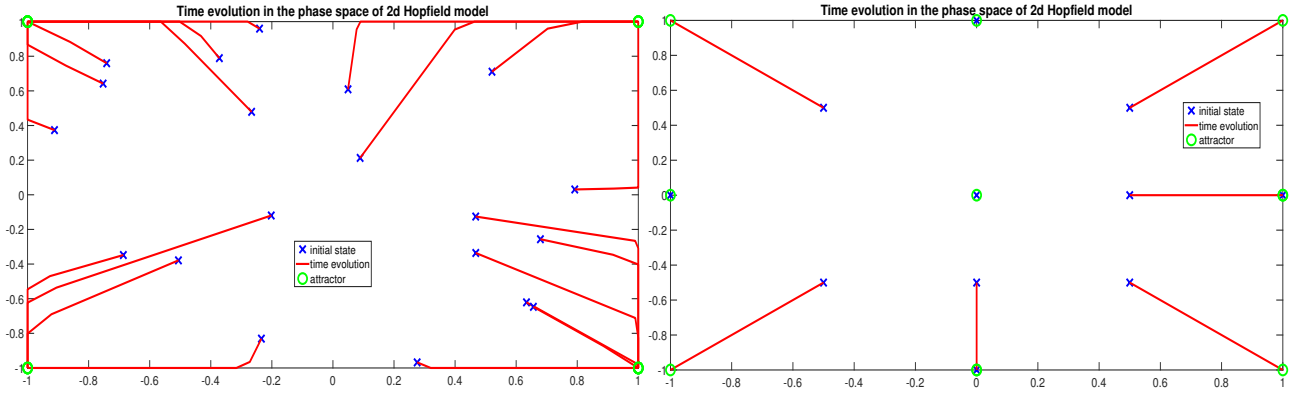
2. Recurrent neural networks

2.1 Hopfield Network

A hopfield network is like associative memory where it can reconstruct the input data even if the data is noisy. The network is a chain of neurons where each neuron can take either of one state as +1 or -1. Suppose for a neuron i at given time t, the state of the neuron can be expressed as $S_i(t) = +1$ or $S_i(t) = -1$. The dynamics of the network evolves in discrete time steps Δt which means the output of a neuron at time t becomes an input at time $t + 1$. Each neuron interacts with each other with a weight matrix w_{ij}. The action potential of a neuron i due to the activity of $i - 1$ neurons can be given as follows

$$h_i(t) = \sum_j w_{ij} S_j(t) \tag{2.1}$$

To demonstrate the working of the hopfield network, we create the network with 3 attractors $T = [1\ 1; -1\ -1; 1\ -1]'$ in two different scenarios. One scenarios where we randomly assign the initial states for the network and the other scenario where we assign symmetrical states. The result of these scenarios is shown in the below figure 2.1. In figure 2.1a it can be seen that each initial state reaches a attactor that closer to it. It exhibits a nearest neighbour behaviour. One thing that can be noticed is that the number of attractors is not the same as how it has



(a) 2 Neuron: Hopfield network with random initial states

(b) 2 Neuron: Hopfield network with symmetrical initial states

Figure 2.1: Time evolution in the phase space of 2D Hopfield model

been created. The attractors are more in number amd can be seen as green circles in figure 2.1. The networks adds a new attractor at (1,1) known as spurious state in figure 2.1a. Spurious states are a retrieval states that the network uses during the pattern training, which eventually becomes an attractor. The energy in these spurious states is also a local minima.

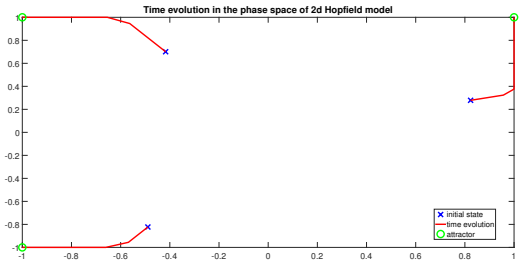
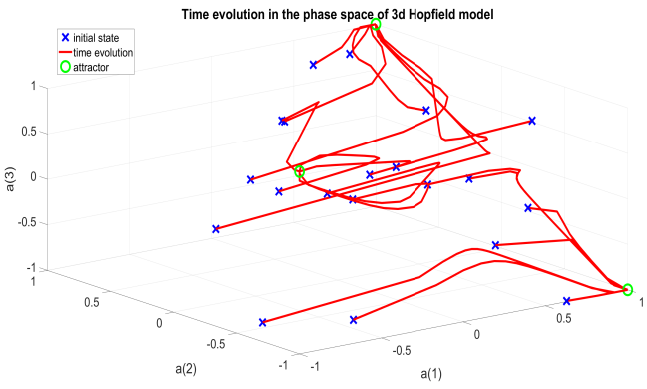
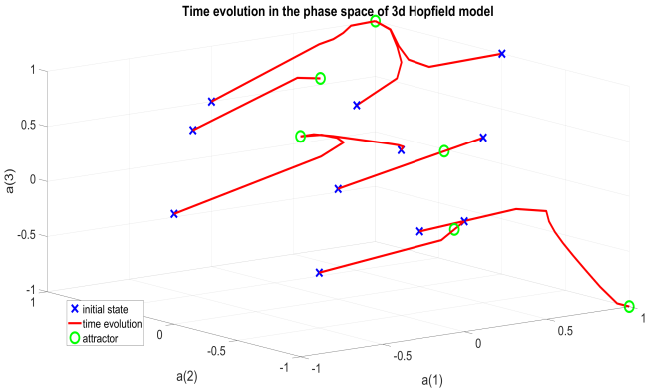


Figure 2.2: Hopfield network convergence to an attractor

Repeated updates to the network states leads to the convergence to a retrieval state which is a spurious state. In the case of symmetrical states as shown in figure 2.1b, the network adds more spurious states and during convergence some states become attractors themselves. The number of attractors for symmetrical points are more than random ones. The network takes atleast 15 iterations to converge to an attractor in case of 3 random initial states that is shown in figure 2.2. In case of a 3 neuron hopfield network, the network correctly converges to the attractors for randomly initialized states as seen in figure 2.3a. But, for symmetrical states, the network produces spurious attractors as seen in figure 2.3b



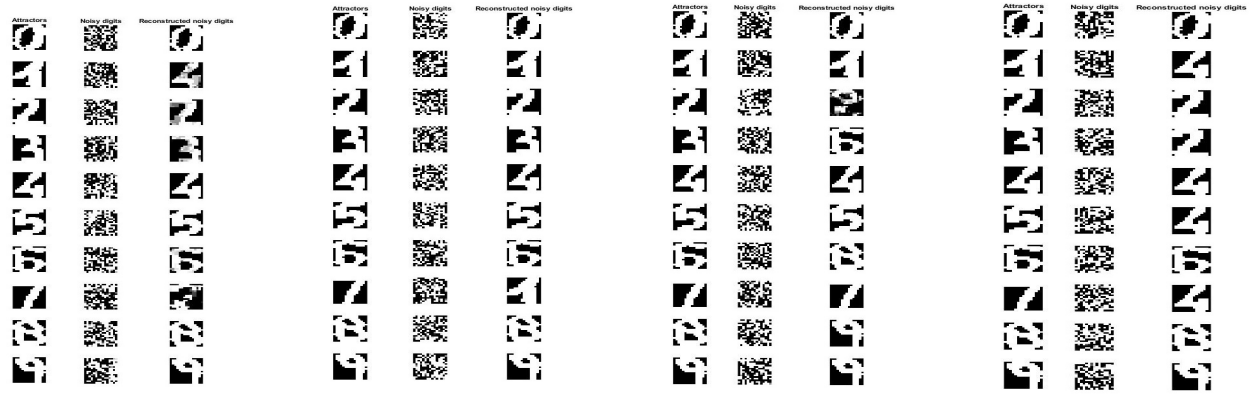
(a) 3 Neuron: Hopfield network with random initial states



(b) 3 Neuron: Hopfield network with symmetrical initial states

Figure 2.3: Time evolution in the phase space of 3D Hopfield model

We then move on to reconstructing the hand written digits using *hopdigit* function. The network is trained with proper handwritten digits which acts as attractors and then inputting different noise levels over the handwritten digits as initial states for reconstruction. The network is tested with four different configurations of noise levels and iterations. The network's behaviour is shown in the figure 2.4. It can be noticed that, for lower levels of noise (say 5) the network is able to reconstruct the pattern correctly to some extent except for one digit. For lighter levels of noise, the networks performance reduces further even after giving higher number of iterations. This could be because the network would have converged to a spurious state which is also a local minima and can be seen in figure 2.4c of digit 2 with Noise level 10 and iterations 50 .



(a) Noise Level = 5; Iterations = 10 (b) Noise Level = 5; Iterations = 80 (c) Noise Level = 10; Iterations = 50 (d) Noise Level = 10; Iterations = 100

Figure 2.4: Hopfield Network for handwritten digit recognition

2.2 Long Short-Term Memory Networks

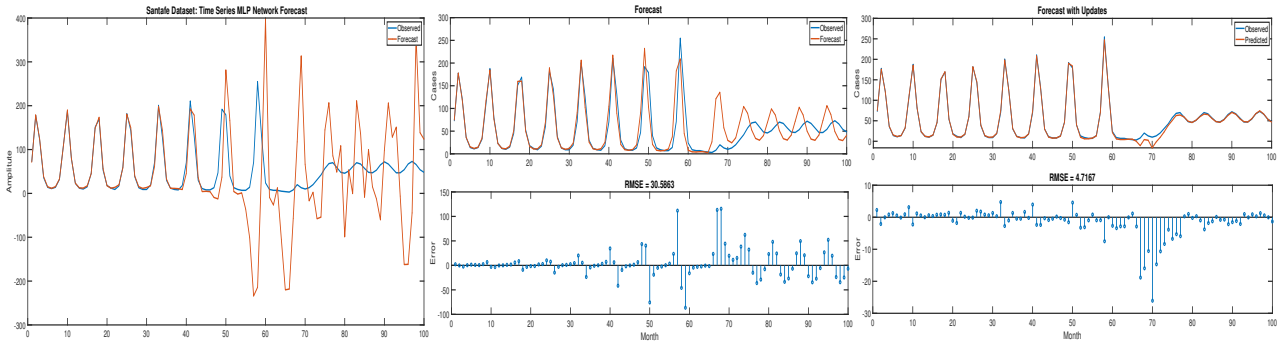
In this session we will train a recurrent multi-layer perceptron (MLP) and a LSTM network on Santafe dataset. The MLP network has limited capability of predicting a sample for only one step in the future. For this purpose, we iterate the network prediction for every step to predict one step into the future. We train this network using feed-forward with back propagation. On the other hand, LSTM networks are built to overcome the issue of long term dependencies which a normal recurrent neural network fails to learn. The LSTM network is made of a special memory cells which consists of gates. These gates learn and control the information flow in the network. They decide on if an information is to be passed on to further nodes or to be thrown away. Therefore, during training the network learn three different weight matrices namely W_i corresponding to input gate, W_f for forget gate and W_o for output gate. The three gates can be characterized by the following equations Input Gate; $I(t) = \sigma(W_i x(t) + U_i h(t–1) + b_i)$, Forget Gate; $F(t) = \sigma(W_f x(t) + U_f h(t–1) + b_f)$ and Output Gate; $O(t) = \sigma(W_o x(t) + U_o h(t–1) + b_o)$, where $h(t-1)$ is a hidden state at time $t-1$ and $x(t)$ is the input at time t.

2.2.1 Time-series Prediction and Long short-term memory network

Time series prediction using a MLP is explored with on hidden layers and various hidden units. The model is trained with different lags on the data sets to evaluate its performance because lags can change the number of training samples and training features. Therefore, the model is evaluated for lags in the range of 10 to 90 and (50; 100; 200) for the number of hidden units in a layer. The network is trained for 500 epochs for each of these hyper parameters and their corresponding RSME values are shown in the below figures. From figure 2.5a, it can be seen that for the first part of the test set where there are smaller spikes, the model performs better and gives a good prediction. But in the second part where there are huge spikes, the recurrent MLP failes to give a good prediction. The figure shown here is with a network having 100 hidden units in a layer. The overall performance of this network is not that great as it has

an RMSE of 56.32 (%). One major disadvantage of recurrent neural networks is that they suffer from vanishing gradients during back propagation. This can be over come by using LSTM's.

As seen from figures 2.5b and 2.5c, the LSTM network predicts better than the MLP network on the test set. The RMSE values of LSTM network without the updates is 30.5863 which is still a higher value but smaller compared to the MLP network. When updates are taken into consideration, the RMSE value further reduces to 4.7167 which indicates that LSTM out performs the ordinary recurrent neural network.



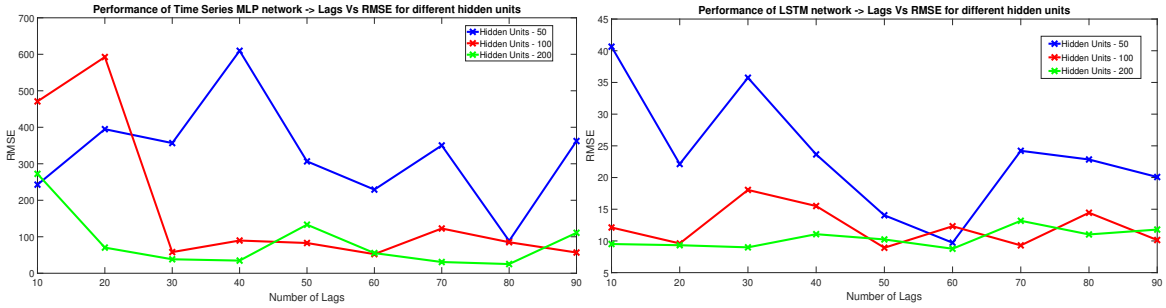
(a) MLP Network; Actual and Predicted output on test data

(b) LSTM Network; Actual and Predicted output on test data

(c) LSTM Network; Actual and Predicted output with updates on test data

Figure 2.5: Santafe Dataset; Actual and predicted outputs on test data with RMSE for MLP and LSTM networks

The figure 2.6 shows the performances of both the networks in case of change in lags and number of hidden units. It can be seen that the RMSE values over the number of lags for LSTM network is almost 10x times smaller compared to the recurrent MLP network. Therefore, using a LSTM network would be a better choice for all the above said reasons.



(a) MLP Network; Number of Lags Vs RMSE for (50, 100, 200) hidden units

(b) LSTM Network; Number of Lags Vs RMSE for (50, 100, 200) hidden units

Figure 2.6: Santafe Dataset; Number of Lags Vs RMSE for MLP and LSTM networks

3. Deep feature learning

3.1 Principal Component Analysis

PCA is a dimensionality reduction technique where data observations of N dimensions are projected onto a lower dimension space. PCA involves computing the covariance matrix of the normalized input vector which gives the information about the principal components (eigen vectors). These eigen vectors defines the amount of variance in each principal component. In this exercise, we first apply PCA on two different datasets. One is randomly generated Gaussian data of 500 observations and 50 dimensions and the other is choles data with 264 observations and 21 dimensions. The results of PCA on both these data is shown in the figure 3.1.

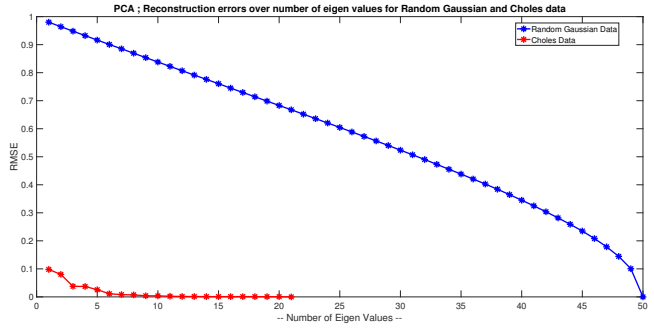
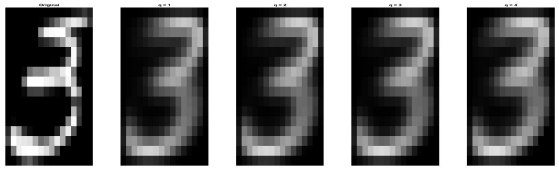
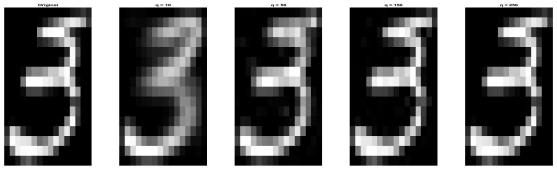


Figure 3.1: PCA on random Gaussian and Choles datasets

From the figure 3.1 it can be noticed that for highly co-related choles data, the reconstruction error is very minimum compared to randomly distributed gaussian data. This is because, the variance among the data observations are very less compared to the random gaussian data. Also, the reconstruction error becomes close to zero after few eigen values for choles data whereas random gaussian data takes 50 which is its original dimensions. We then proceed to perform PCA on handwritten digits dataset of digit 3 for first 4 principal components and then increase the number of principal components upto 256. The results are shown in the below figure 3.2. It can be noticed from the figure that for higher



(a) Original image and reconstructed images with PCA Components [1, 2, 3, 4] - second image to end from left



(b) Original image and reconstructed images with PCA Components [10, 50, 150, 256] - second image to end from left

Figure 3.2: PCA on handwritten digit images of digit 3

values of principal components, the reconstruction becomes more clear as almost all the information is retained. For p=256, we can see that the reconstruction error is zero, as it is the same value as the dimensions of the input data.

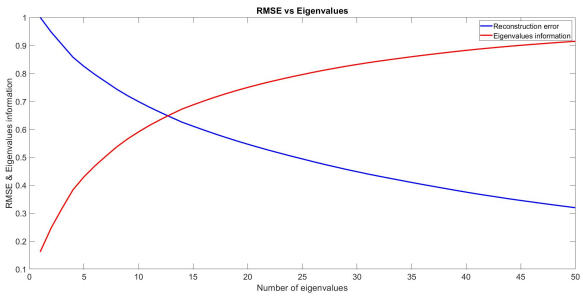


Figure 3.3: PCA: Reconstruction errors VS Eigen value information

We then try to analyze the relationship between the reconstruction errors and eigen value information. The plot shown here is for normalized values of errors and cumulative sum of eigen values. From figure 3.3 it can be noticed that the errors and eigen values have an inverse relationship.

3.2 Stacked Autoencoders

A stacked encoder is a neural network with a group of sparse auto encoders where the output of each auto encoder is connected to the consecutive ones. These encoders are trained layer-wise in unsupervised fashion. The output layer of the last encoder can be connected to a softmax layer for the task of classification. We train the stacked encoder for digit classification and some of its results along with the results of a normal neural network is given in the below table.

(a) Stacked Auto-encoder

Hidden Layer (HL1)	Hidden Layer (HL2)	Epoch 1(HL1)	Epoch 2(HL2)	Accuracy (%)
100	50	400	100	99.66
100	50	600	200	99
100	75	400	100	99.78
150	75	400	100	95.76
150	30	400	100	99.70
200	50	400	100	95.42

(b) Neural Network Model

Hidden Layers	Accuracy(%)
1 Layer [100]	97.68
2 Layers [100 50]	96.98
1 Layer [150]	96.80
2 Layers [150 100]	95.24
1 Layer [250]	97.26
2 Layers [150 30]	88.72

Table 3.1: Performance of stacked autoencoder and neural network models

The results showed in the table 3.2a are obtained by tuning the hyper parameters; number of epochs in hidden layers 1, 2 and number of hidden units in each layer. The stacked auto-encoder with 100 hidden units in hidden layer 1 and 75 hidden units in layer 2 gives better accuracy than other models. Fine tuning has a huge influence on the classification accuracy as the model learns better weights from the encoder layers. On the other hand neural network model does'nt perform as much as the stacked auto-encoder since, the neural network model is trained as a complete network and the encoder is trained layer wise.

3.3 Convolutional Neural Networks

MLP's are not suitable for all neural network applications. MLP's are not translation invariant and using them for object recognition, classification or any other computer vision tasks can be a bad idea. Convolution neural networks shorty knows as CNN's are popular in most of the computer vision with deep learning tasks. These CNN's are made of convolution blocks that repeatedly applies a filter over the input image to get activation known as feature maps. These features are learnt over various layers of the convolution network and these feature map information is the base for any object detection or classification. In this exercise we first run the *CNNex* script to analyze the working of a CNN. The script uses AlexNet architecture for a classification task on three different classes of objects aeroplane, laptop and a ferry from Caltech101 dataset. The first layer of AlexNet is a input layer that takes an input image of size 224x224x3. The first two layers of the networks uses a max pool layer and the subsequent three layers are connected directly followed by another max pool layer after fifth conv layer. The final three layers are fully connected layers and a softmax layers with 1000 output nodes that gives out probability of the object in a image belonging to a class.

Between every convolution layers there is a ReLU layer that only activates positive inputs and a Normalization layer that normalizes the previous layer output with its mean and standard deviation. The output dimensions of every convolution layer is given by $[\frac{(n+2p-f)}{s} + 1]\ X[\frac{(n+2p-f)}{s} + 1]\ Xn_c$, where n is size of the input, s is stride, f is filter and n_c is number of ouput channels. The below figure shows the feature map of the first layer of the convolution network. From the figure 3.4a, the weights shown for the first layer can learn basic features like

(a) CNN : AlexNet convolution layer 1 weights (b) Caltech101 dataset images

Figure 3.4: Visualization of CNN weights and datasets

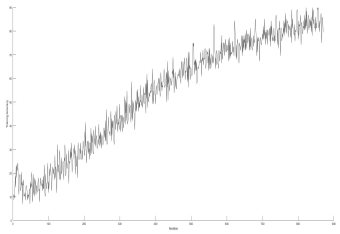
edges, strokes, blobs etc. As we go deeper into the layers better features such as shapes are learnt. The first convolution layer in the network uses a filter of size 11x11x3, a stride of 4 and 0 padding layers. Therefore using the output formula as mentioned above we get the output size of first convolution layer as 55x55x96. ReLU and normalization layers do not alter the output dimensions. The output after the max pool layer is given by $[\frac{n-P_s}{S} + 1]$, where P_s is the pool size. The network uses a max pool size of 3x3 and a stride of 2. Therefore, the size of the layer

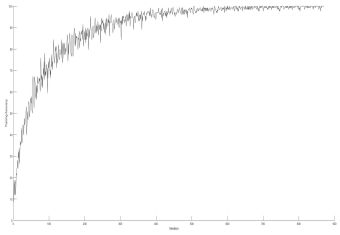
after first max pooling is given by (55-3/2)+1 = 27x27x96. Similarly, the output after fifth convolution layer and max pooling is obtained as 6x6x256. The final fully connected layers have a size of 4096 and the softmax layer as 1000(number of classes in our case) compared to input dimensions 227x227x227=154,587 as the image size gets down sampled over the convolution and max pool layers.

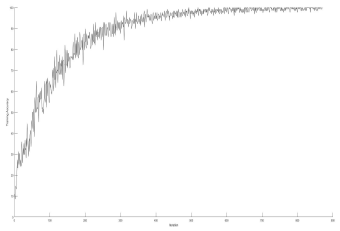
CNN's are better at feature extractions than fully connected layers. The number of weights required for a CNN training gets reduced by down sampling over the convolution layers which is not the case with only fully connected networks. Due to the large number of parameters to be tuned in the network, FCN's can take longer durations for training and chances of over-fitting is also high.

We then run the script *CNNDigits* to evaluate the classification capabilities of a CNN. We train and test the models by changing various layers. Some of the results for the same are shown in the table 3.3. The trained is done on GPU, therefore the training time for all the architectures that are experimented is less than 40 secs. It can be noticed that the architecture with convolution layer1 with 32 filters and size 5, convolution layer2 with 64 filters and size 5 gives maximum accuracy of 98.24. The architecture with 4 convolution layers and filter size 3 couldn't give better performance than the one with 2 layers and filter size 5. From these results it can be said that the choice of convolution layers and filters influence a CNN's performance.

CNN Architecture Layers	Training Accuracy Graph	Training time and accuracy

convolution2dLayer(5,12);
reluLayer;
maxPooling2dLayer(2,'Stride',2);
convolution2dLayer(5,24);
reluLayer
		Accuracy = 82.08
		Time elapsed = 25.611191 secs

convolution2dLayer(3,32);
reluLayer ;
maxPooling2dLayer(2,'Stride',2);
convolution2dLayer(3,64);
reluLayer;
maxPooling2dLayer(2,'Stride',2);
convolution2dLayer(3,128);
reluLayer;
convolution2dLayer(3,256);
reluLayer
		Accuracy = 97.60
		Time elapsed = 37.527818 secs

convolution2dLayer(5,32);
reluLayer;
maxPooling2dLayer(2,'Stride',2);
convolution2dLayer(5,64);
reluLayer
		Accuracy = 98.24
		Time elapsed = 33.306200 secs

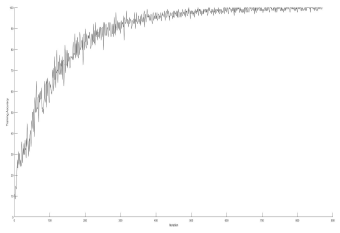
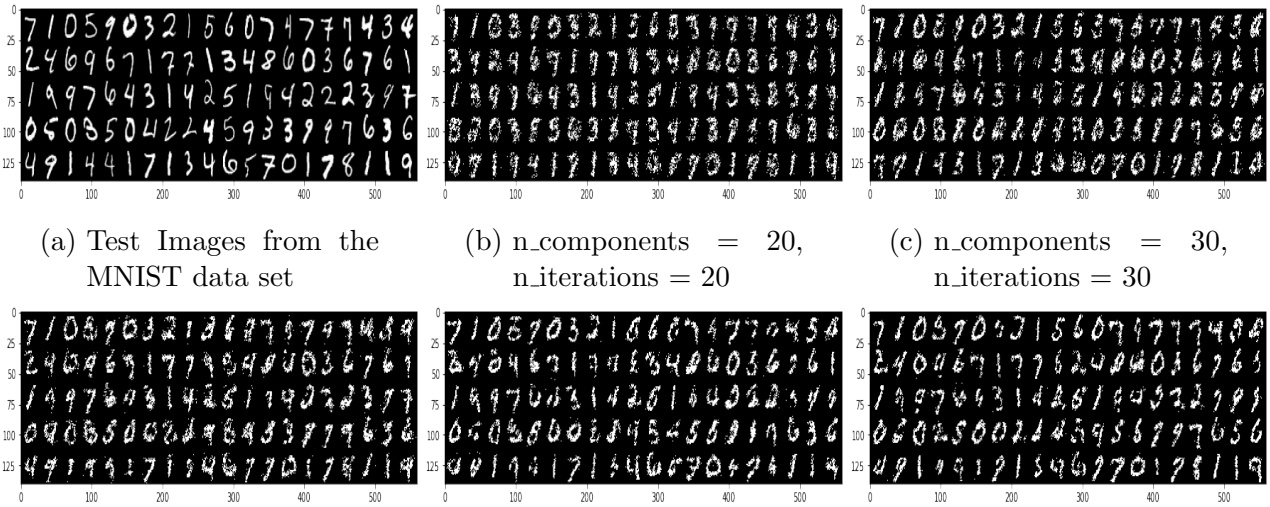
convolution2dLayer(7,32);
reluLayer ;
maxPooling2dLayer(2,'Stride',2);
convolution2dLayer(7,64);
reluLayer;
		Accuracy = 97.68
		Time elapsed = 27.847668 secs

Table 3.3: Performance of a CNN in classifying digits dataset for various architectures

4. Generative models

4.1 Restricted Boltzmann Machines

The Boltzmann Machines come under the category of Energy Based Models. The name energy based models is given because they make use of Boltzmann distribution also known as Gibbs distribution which is a part of statistical models that helps in determining the quantum state parameters like Entropy, Temperature etc. These Botlzmann models are stochastic generative deep learning models that have only two nodes namely hidden and visible nodes. Like conventional neural network models, they explicitly do have output nodes with 0 or 1 states through which they learns the patterns. Each of the nodes of these models are linked to one another irrespective of they type (either input or hidden) and do they learn and share the information among these nodes. This makes them non-deterministic and since they self generate the subsequent data, they are known as deep generative models. A Restricted Boltzmann machine (RBM) is a generative two-layered neural network model with restrictions in its hidden and visible nodes. They have the capability to learn a probability distribution given a set of observations. In this exercise, we evaluate the performance of a RBM on MNIST digits dataset. We train the RBM with different parameters like number of iterations, number of components, gibbs sampling steps etc. The model is then evaluated on unseen images for each of the configuration. To first analyze the effect of training parameters like number of components, number of iterations etc. we keep the gibbs sampling steps to constant value of 20, learning rate to 0.01, batch size to 10 and change the training parameters and the results are as given below.



(a) Test Images from the MNIST data set

(b) n_components = 20, n_iterations = 20

(c) n_components = 30, n_iterations = 30

(d) n_components = 40, n_iterations = 40

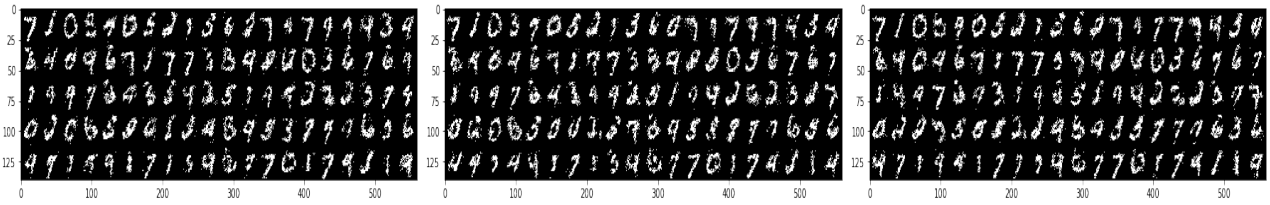
(e) n_components = 50, n_iterations = 50

(f) n_components = 60, n_iterations = 60

Figure 4.1: Gibbs sampling on test images for different train parameter configurations

From the model evaluation results on the test set it was observed that, change in any of these

n_components and n_iterations parameters have the same impact on the model performance. Increasing these parameters increase the pseudo-likelihoods. For higher values of these parameters, the time taken for each iteration is higher. As seen from figure 4.1, the model performance for higher values of n_components and n_iterations is better compared to 20, 30 and 40. Except for the digits 2, 4 and 5, the model in figure 4.1f does a good job in giving out better sampled images. Now, to evaluate gibbs sampling steps, we fix the train parameters n_components and n_iterations to 40, learning rate to 0.01, batch size to 10 and vary the gibbs sampling steps. Few results are shown below.



(a) Gibbs sampling steps = 30 (b) Gibbs sampling steps = 40 (c) Gibbs sampling steps = 50

Figure 4.2: Gibbs sampling on test images for different Gibbs sampling steps and constant train parameter configurations

From the figure 4.2 it can be seen that there is some improvement in the model performance for increase in gibbs sampling steps but not to a great extent.

Reconstructing Missing parts : It is observed that, increasing the number of components,

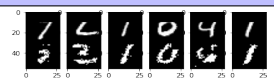
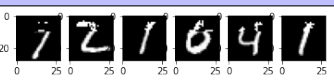
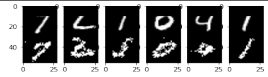
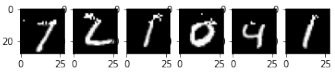
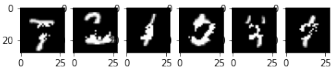
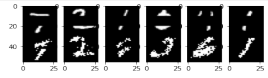
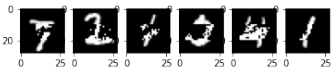
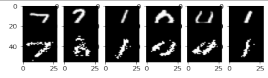
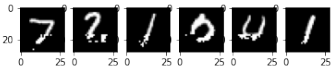
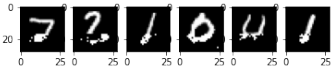
Rows to be Removed	Gibbs Sampling Steps	Sampled Image with missing parts	Reconstructed Image
Top rows 1-10	30		
Top rows 1-10	50		
Middle rows 10-20	30		
Middle rows 10-20	50		
End rows 17-27	30		
End rows 17-27	50		

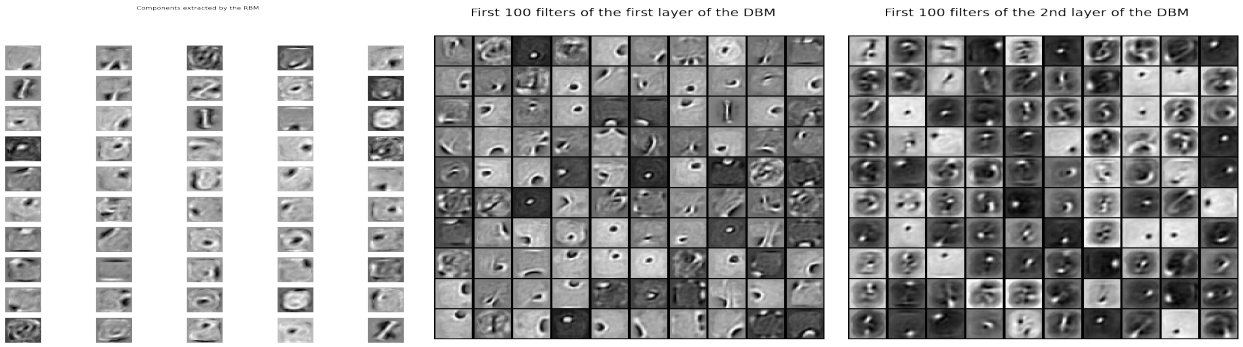
Table 4.1: RBM performance for reconstructing missing parts of image with model parameters as n_components=50, n_iterations=50, learning rate = 0.01, batch size = 10

number of epochs and gibbs sampling steps the model makes reconstruct better. Reducing or

increasing the learning rate has poor performance on the model output. The table 4.1 summarizes the results of a RBM in reconstructing the images. It can be seen that the model gives better results when top rows are removed compared to middle and end rows.

4.2 Deep Boltzmann Machines

Deep Boltzmann Machines (DBM's) are nothing but RBM's stacked on one another. The features extracted from one layer of DBM is passed to the second layer of DBM. By doing this the model learns the image features in a better way. The figure shows the features of RBM with 50 components (hidden units) from the above exercise and the feature of 2 layers of DBM.



(a) Feature map of RBM from previous section

(b) Feature Map of 1st layer of DBM

(c) Feature Map of 2nd layer of DBM

Figure 4.3: Features maps of DBM and RBM

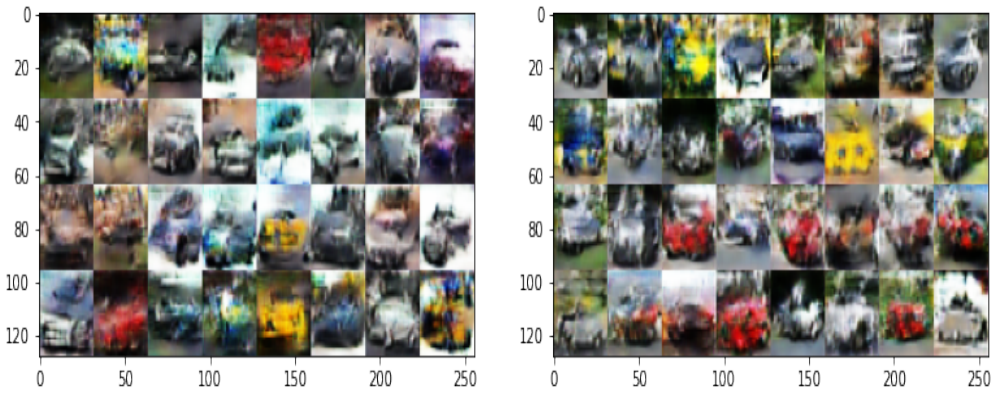
Figure 4.4: DBM samples after 100 Gibbs steps

The figure 4.3b shows layer 1 basic features of DBM which are extracted from the layer 1 filters. Figure 2 shows the layer 2 edge features of DBM. It is evident that the feature maps of DBM is better than RBM as DBM has more filters as compared with RBM with 50 components as shown in figure 4.3a. The deeper the network, the better the extracted input features are and so is the model's performance. Since the DBM model has better feature maps with 2 layers and more number of hidden units than a RBM model, the result of DBM shown in figure 4.4 is more clear and accurate than a RBM model.

4.3 Generative Adversarial Networks

As in their name, GAN's can generate new synthetic data using two different neural networks namely the generator and the discriminator. The generator generates a new instance while the

discriminator tries to distinguish the generated data from the ones in the training set and assigns a probability to every new data generation that if a sample corresponds to any from training set or a fake. This way both the neural networks compete in a zero-sum game trying to maximize their performance. In this exercise, we analyze the GAN's using Cifar dataset. We train the GAN's for 15000 batches and notice that the model generates car images with D-loss : 0.6914 D-acc : 0.5136 G-loss : 0.6746 and G-acc : 0.3147. The generated images seems to be of poor resolution and they are kind of blurry. May be this output could be improved by increasing the batch size as well as number of epochs. Some of the generated images for batches 10000 and 15000 are given in the below figure

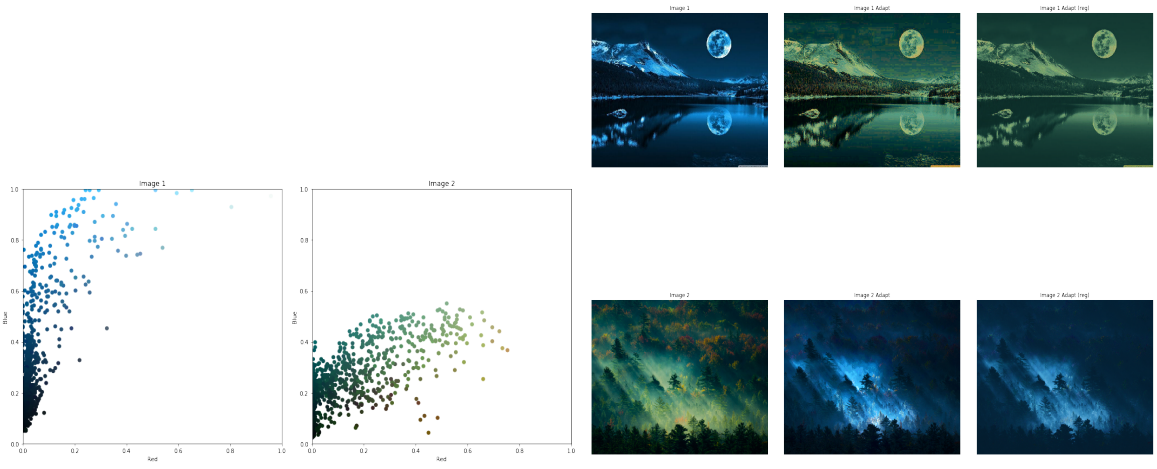


(a) After 10000 batches (b) After 15000 batches

Figure 4.5: GAN image generation

4.4 Optimal transport

This exercise deals with color transport between two images where the color palette of one image is imposed on the other. This means, after the color transport the color distribution of the resultant image will be very close to the input image. For this purpose we use two distance metrics namely Wasserstein metrics and Sinkhorn metrics. The transport of the color distribution is done using these distance which is a measure of work done i.e., to transport the mass of one distribution to another. Figure 4.6a shows the color distribution scatters of the two images. Figure 4.6b shows the resultants of these color transports. The first column of images in figure 4.6b are the input images. The second column shows the optimal transport using Wasserstein distance and the third column corresponds to Sinkhorn distances. It can be seen from these images that using Wasserstein distance makes the output images very sharp but they can preserve minute details of the original images. On the other hand Sinkhorn distance does a better job by giving a smooth and clear output due to the fact that it uses entropy to regularize the two distributions.



(a) Color scatters of image 1 and image 2 (b) Color Swapped Images

Figure 4.6: Optimal color transport between two images

We then compare a standard GAN with Wasserstein GAN by training and evaluating them on MNIST dataset. We train each of the models with the following parameter configuration; batches = 20000 , batch size=64 and plot interval = 500. The results of these GAN's are shown in the table 4.2 for 1000, 10000 and 20000 batches of training. It was observed that, a standard GAN takes lesser time in training around 65 iteration per seconds. At the end of 20000 iterations we get the loss and accuracies of the discriminator and generator as D-loss: 0.5373 D-acc: 0.7344 G-loss: 1.3259 G-acc: 0.125. The generated images are not of high quality and accuracy. They also contain salt and pepper noise as seen for 1st row of table 4.2. These GAN's suffer from weight clipping where all the weights of the GAN's vary only between a range $[min, max]$ values. Due to this they cannot model complex structures as their learning is restricted. To overcome this issue Wasserstein GAN's.

In Wasserstein GAN with weight clipping, the GAN uses a clipping that restricts the maximum weight value in its function. This means, the learning is controlled by the clipping hyper parameter. With this the model performance improves a bit as seen from row 2 of table 4.2. The performance of this WGAN with clipping depends on the tuning of the clipping parameter. A smaller value of this could cause vanishing gradients and larger value could cause longer time for the weights to reach their limits. Therefore, we move on to Wasserstein GAN with gradient penalty.

The issues of weight clipping is solved by using a weight penalty that enforces the gradients to have a norm 1 if they move away from their target norm. This method produces some good quality images as seen from row 3 of table 4.2. Training this model takes more time of around 13 iterations per seconds compared to other two models.

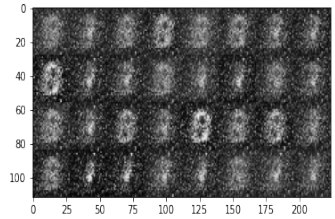
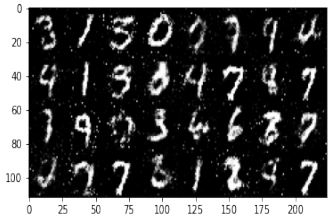
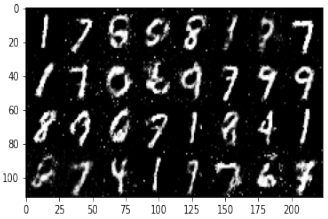
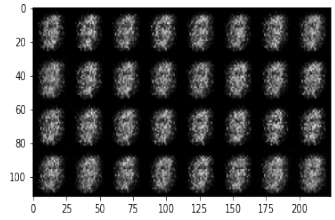
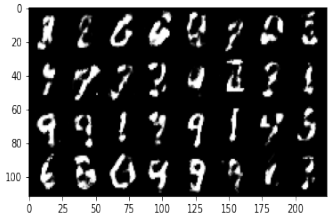
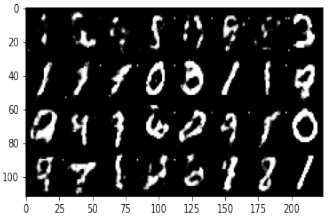
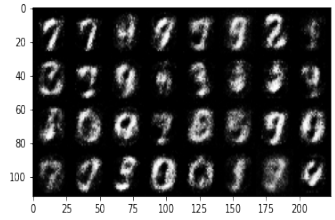
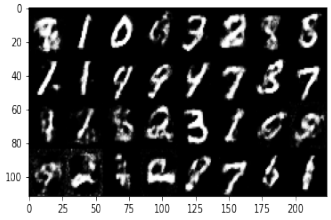
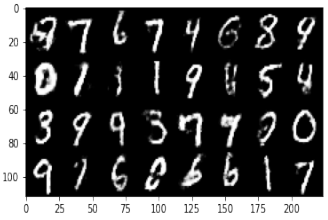
GAN Techniques	GAN output at batch 1000	GAN output at batch 10000	GAN output at batch 20000
Standard GAN			
Wasserstein GAN with Weight Clipping			
Wasserstein GAN with Gradient Penalty			

Table 4.2: Results of standard GAN and WGAN's trained on MNIST dataset