Faculty of Engineering Science
Master in Artificial Intelligence
Academic year 2022-2023

Artificial Neural Networks and Deep Learning (H02C4A)

Supervised by Professor Johan Suykens

Submitted By:

Hussain Abbas (Exchange Student)
Matricola Number: r0909805

Leuven , 30th May 2022

Part I: Supervised Learning and Generalization

1. The Perceptron

a. Function Approximation: Comparison of Various Algorithms

A function $y = \sin(x^2)$ with x ranging from 0 to 3π with steps of 0.05, has been approximated using a feedforward neural network with just one hidden layer with neurons defined by the above transfer function. A plethora of training algorithms are pitted against each other in this section.

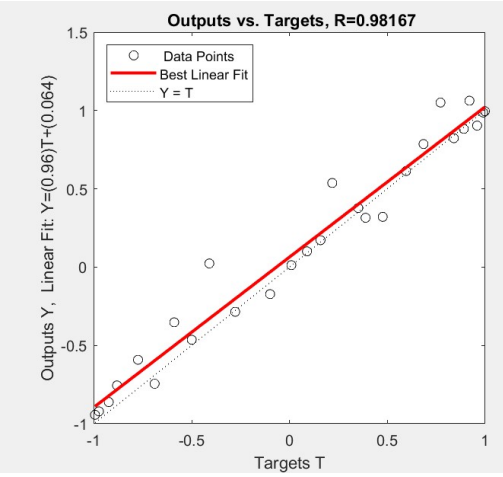


Figure 1: Outputs vs Target

Mean Square Error metric is wielded as the loss function to evaluate the performance of the algorithms as it elucidates how close a regression line is to a set of data points. The figure on the left exhibits the Best Fit regression line after the training phase. The figure encompasses the ideal line (Y=T) and the output line.

Six distinct training algorithms (gradient descent, gradient descent with adaptive learning rate, Fletcher-Reeves conjugate gradient algorithm, Polak-Ribiere conjugate gradient algorithm, BFGS quasi Newton algorithm and Levenberg-Marquardt algorithm) are deployed on the function devoid of noise and are plotted against each other based on three factors namely, Mean Square Error, number of Epochs and net duration of the training phase

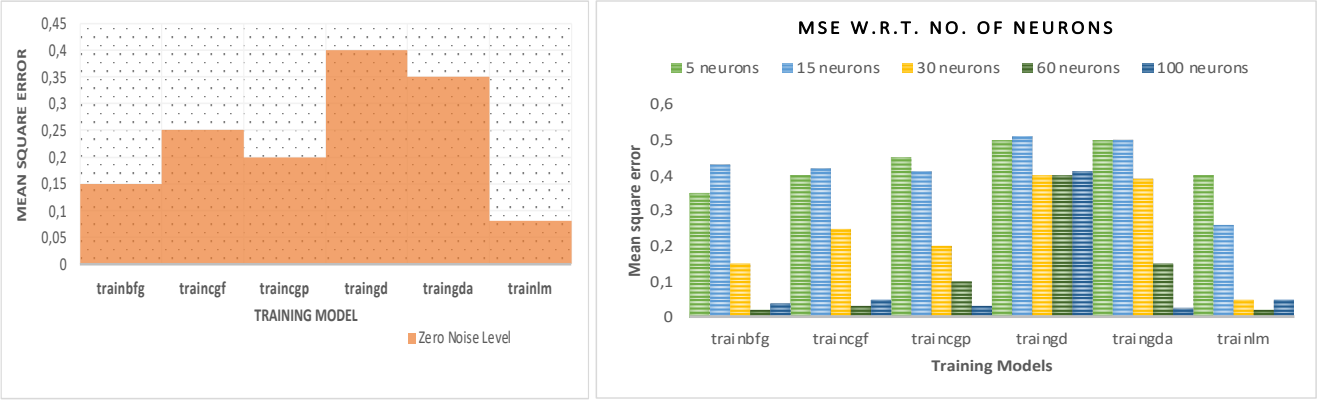
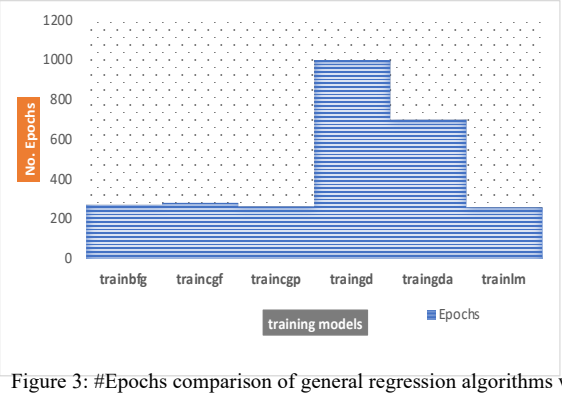


Figure 2: Mean Square Errors with no noise and with respect to number of neurons

The Levenberg-Marquardt algorithm spits out the least Mean Square Error out of all of them. Gradient Descent performs the worst followed by gradient descent with adaptive learning rate. With increase in the number of neurons, an enhancement in performance of all algorithms can be perceived, except for Gradient Descent. With lower number of neurons, all algorithms lay out a high error rate which might be a diametrical consequence of a possible underfitting.



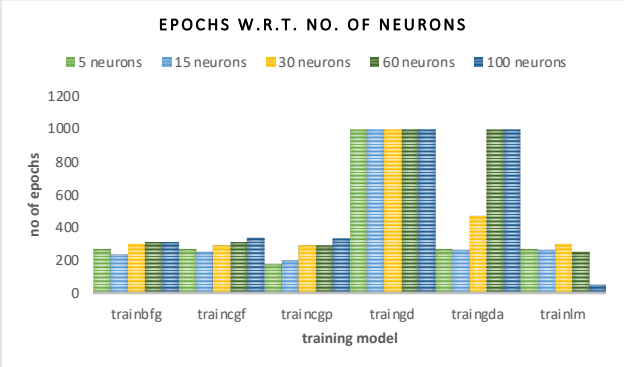
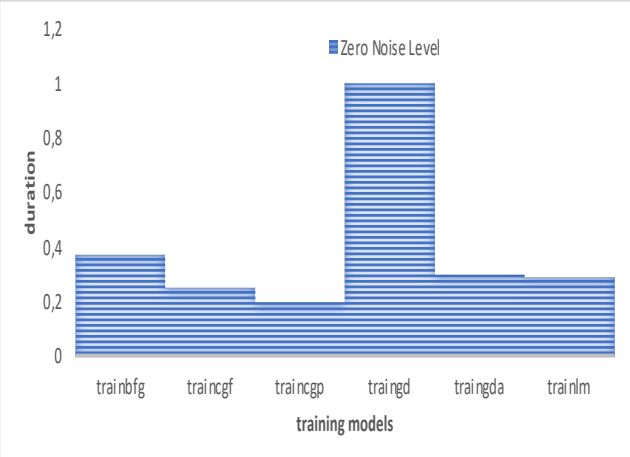


Figure 3: #Epochs comparison of general regression algorithms with no noise with respect to number of neurons

Gradient Descent expends the maximum number of Epochs and the Levenberg-Marquardt algorithm squanders the least. The LM algorithm unlike any other algorithm avails lesser Epochs with an escalating number of neurons. The first rationalization that may come to one's mind is overfitting. But overfitting is typically not a characteristic of an algorithm, it's instead a repercussion of the way the data is setup. The number of neurons in the hidden layer is proportional to the degree of freedom in the input space. All the algorithms except LM, come across as over-trained, and as a result, must have "memorized" the input and must not have performed any generalization.

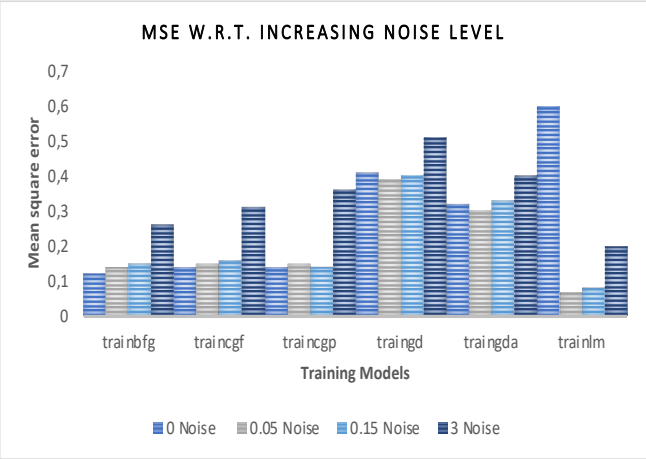


Gradient Descent burns through for the maximum duration (average) during training while and the conjugate gradient algorithms are the quickest. Gradient descent iteratively searches for a minimizer while being gradient directional, but conjugate gradient calls for the need to search in a direction orthogonal to each other. All algorithms spend more time with an increase in the number of neurons, however results on Gradient Descent do not feature a Low to High shift in the duration as it exhausts a significantly higher time with lower number of neurons when compared to other algorithms. LM manages to keep the time factor low enough throughout the changes in the number of neurons.

Figure 4: Duration comparison of general regression algorithms with no noise with respect to number of neurons

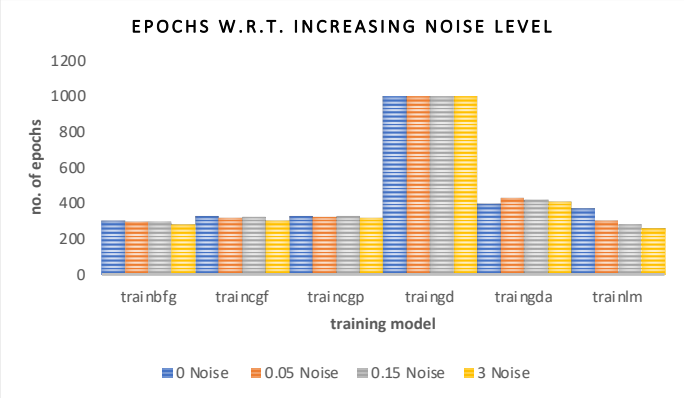
b. Learning from Noisy Data

The performance of the network in the previous section is contrasted with the performance of the network with added noise. Diverse levels of noise are incorporated in the training data and the network is trained using the algorithms seen in the previous section.



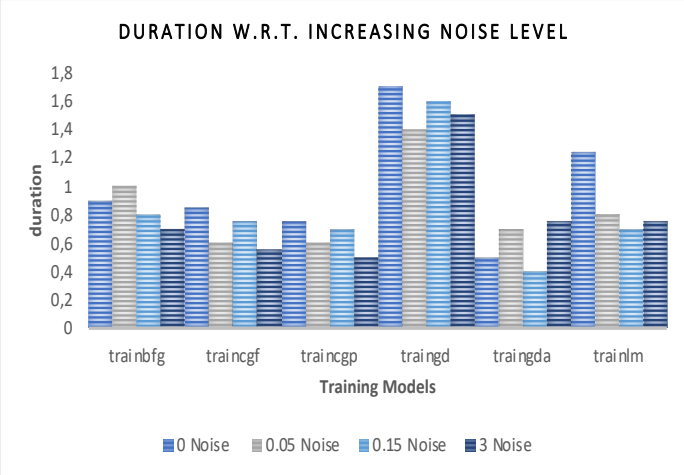
Gradient Descent incurs the worst performance when compared to the errors attained by the other algorithms. As expected, the error increases with the increasing noise level. But overall, LM does a pretty decent job in keeping the error the lowest amongst all the algorithms.

Figure 5: Mean Square Errors with increasing noise level



With increasing level of noise, most algorithms don't boast of a significant increase/decrease in Epoch consumption. The Epoch consumption by Gradient Descent remains same to the effect of the intensifying noise while that of LM decreases with the noise swell. The number of Epochs exploited by each of the algorithms soars with a surge in number of neurons and number of samples. The only exception that can be observed is the dip in Epoch consumption by LM algorithm post 30 neurons. Epochs consumed by Gradient Descent remains unstirred by the change in number of neurons or change in data set size.

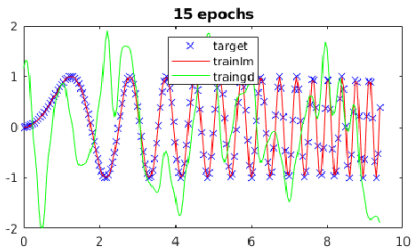
Figure 6: #Epochs comparison with increasing noise level



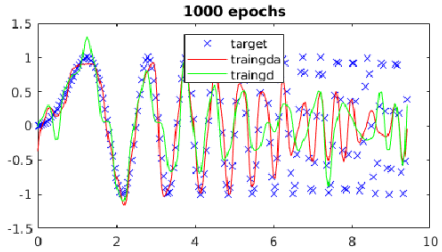
With a build-up of noise, the duration of training remains constant more or less, but with LM being an exception. The duration of training with LM drops with an increase in noise level. As anticipated, the duration of training regardless of the algorithm deployed, increases with a surge in the number of neurons and the size of the training data. Gradient Descent spends the most time on an average when compared to the time spent by the other algorithms.

Figure 7: Duration comparison with increasing noise level

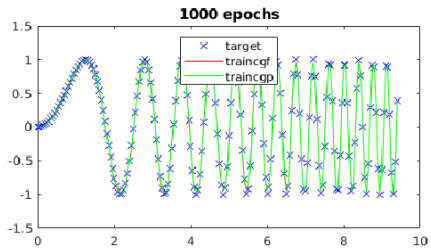
Levenberg-Marquardt vs GD



GD with adaptive learning rate vs GD



Fletcher-Reeves vs Polak-Ribiere



	Without Noise			With Noise		
MSE	Epoch 1	Epochs 15	Epochs 1000	Epoch 1	Epochs 15	Epochs 1000
Gradient Descent (gd)	368.84%	185.40%	32.75%	572.02%	299.55%	21.27%
GD wit Adaptive Learning	269.16%	176.25%	11.12%	601.91%	169.35%	11.67%
Fletcher-Reeves (cgf)	206.56%	23.97%	0.09%	372.37%	19.77%	0.56%
Polak-Ribiere (cgp)	237.89%	19.13%	0.10%	249.84%	41.75%	0.59%
Quasi Newton (bfg)	156.63%	11.30%	0.03%	314.56%	8.22%	0.70%
Levenberg-Marquardt (lm)	11.59%	0.01%	1.90E-09	12.85%	0.59%	0.71%

Table 1. comparison of different training models with respect to no of epochs with and without noise.

2. Personal Regression

A new target is forged representing a nonlinear function and this function is approximated using a feedforward artificial neural network. 13600 datapoints are uniformly sampled from the nonlinear function. The dataset is assembled out of vectors containing input variables in the domain [0,1]X[0,1], and the target vector (built out of my student number – 0909805). Three independent samples of 1000 points each is exploited as the training, validation and test sets. This is accomplished by generating a random permutation of integers from 0 to the size of the column of the X matrix constructed out of the two input vectors.

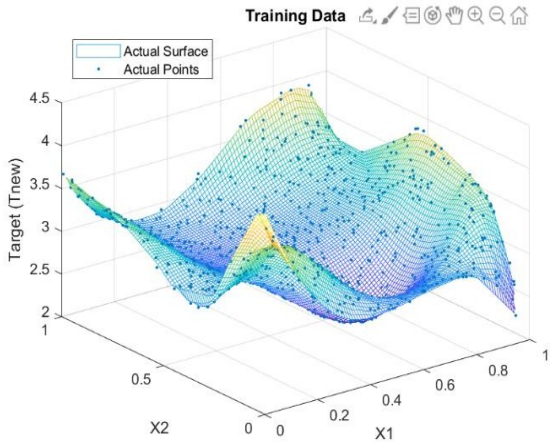


Figure 8: Surface of Training data

The 3D plot on the left exhibits the surface of the training set based on the X matrix containing X1 and X2, and the target vector. An ANN is modelled using two inputs, one hidden layer and one output layer (always linear). The aspects of the network investigated before zeroing into the one best suited for this task, are the number of neurons in the hidden layer, the type of transfer function and the learning algorithm.

The learning algorithm is picked from the list of algorithms stated in the previous sections along with the Bayesian regularization backpropagation algorithm which updates the weight and bias values according to Levenberg-Marquardt optimization.

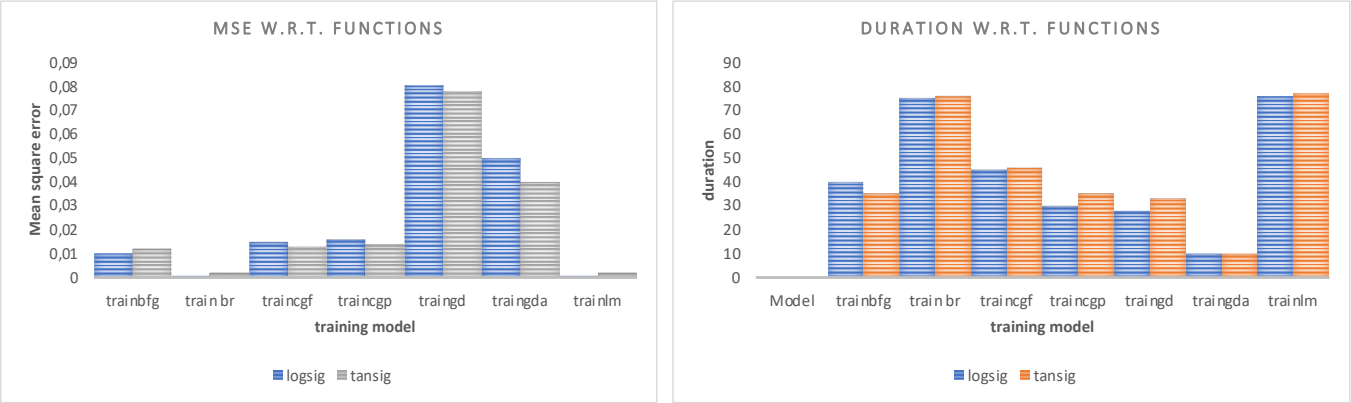


Figure 14: Mean square errors (left) and Duration (right) comparison with respect to logsig and tansig transfer function

a. Best Training Model for this Problem

It is very evident from the above plots that both Bayesian Regularization and Levenberg-Marquardt optimization techniques work well on this task. It is a close call between Logsig and Tansig, but Logsig is favoured based on the marginal edge it has on Tansig.

In order to define a suitable model for our problem, we will train the Neural Network, using the training and validation set, different training algorithms, No of hidden layers (1 and 2) and No of hidden units per layer (2,5,10 or 20). Finally, we calculate the mean squared error of the estimated T and the actual T value of the test set. Initially, we start the training with models of 1 hidden layer. We see that the best performance over the test set (MSE close to zero) occurs from the Levenberg-Marquardt with 20 hidden units. By plotting the overall results, we see an improvement on the performance as we increase the number of hidden units in most of the functions, with some exceptions. When we train the neural networks with 2 hidden layers, we notice that the performance in most cases improves when comparing models with the same number of hidden units per layer. The Levenberg-Marquardt algorithm with 10 hidden units per layer produces the smallest error of the test set.

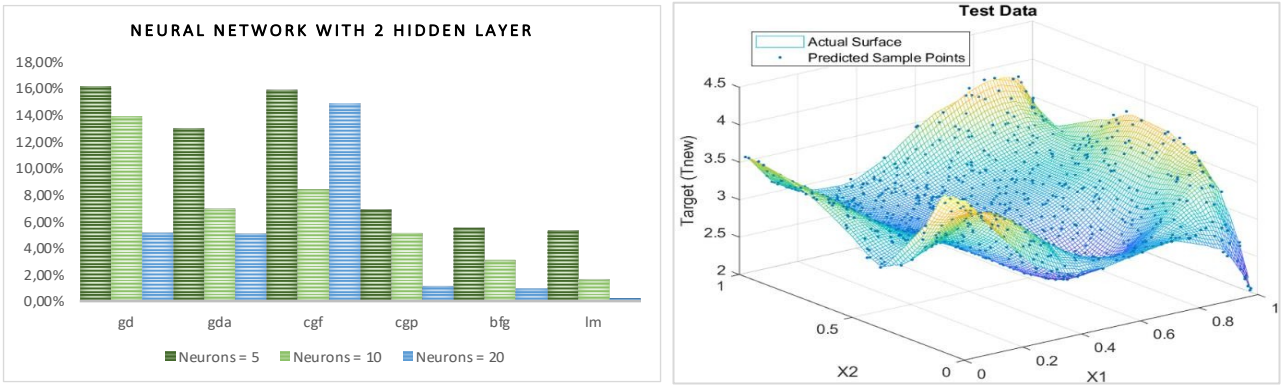


Figure 15: Mean square errors comparison with respect to number of hidden layer*s and predicted sample points compared to the actual surface plot*

Judging from the above results, we will choose the Levenberg-Marquardt algorithm with 1 hidden layer and 20 hidden units. In order to improve the performance further, we could run further combinations of no of hidden units and hidden layers for the specific algorithm and perform Parameter Tuning mechanics or 10-fold Cross Validation. Another method to ensure good generalization of the model is to introduce a regularization term in the objective function, which minimizes the squared weights in order to avoid overfitting. By monitoring regularization, there is a balance between a focus on optimizing the MSE on the training set and ensuring good generalization in unseen data, by avoiding overdependencies on the training set.

3. Bayesian Inference of Network Hyperparameters

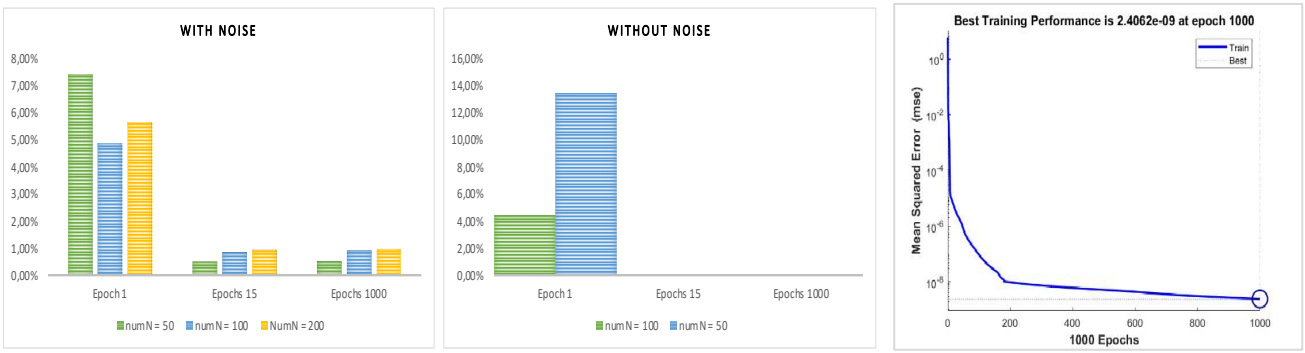


Figure 16: Results of mean square error w.r.t. no. of epochs in Bayesian inference

	Without Noise			With Noise	
Neurons	**numN = 50**	**numN = 100**	**numN = 50**	**numN = 100**	**NumN = 200**
Epoch 1	13.46%	4.45%	7.44%	4.88%	5.62%
Epochs 15	5.00E-06	2.59E-05	0.53%	0.88%	0.96%
Epochs 1000	6.15E-11	7.26E-12	0.55%	0.92%	0.97%

When applying Bayesian Inference as training algorithm and comparing the performance of previous training algorithms with 50 hidden neurons, we notice that the Bayesian framework outperforms the previous top-performer Levenberg-Marquardt. The model with Bayesian framework results to very small errors after only 15 epochs, therefore it is more efficient. On the without noise case, when we tried to train the network with 1000 epochs, the training stopped only after 162 iterations as it converges faster to an optimal minima. In the case with noisy data, the performance is inferior compared to noiseless case, however still outperforms all the rest training algorithms.

Overparameterization of the model with 100 or 200 hidden units is not recommended, as it results in higher errors. This is an effect of overfitting in the model. We need to consider that there are only 189 input data, therefore there needs to be a good balance with the number of parameters we add to our model.The improved performance of the Bayesian Framework compared to other methodologies results from the 3 levels of inference, which optimize the w vector (weights), the hyperparameters a and b of the objective function and model selection. Since all those aspects are included in the framework itself, there is no need to split our dataset in training and validation set, therefore the model can train on the whole dataset, without risk of overfitting or need for early stopping.

Part II: Recurrent Neural Networks

1. Hopfield Network

We start a Hopfield Network with three attractors T and 25 vectors. We run the model for 15 iterations and notice that not all vectors have reached an attractor. When the energy function, which is evolving over time to some lower energy point, reaches an equilibrium at the moment that all vectors have reached one of the available attractors, the training is complete. After increasing the number of iterations to 25, all of the vectors have reached one of the three initial attractors or a fourth attractor added by the model. Spurious states are unwanted attractors that are linear combinations of the stored patterns.

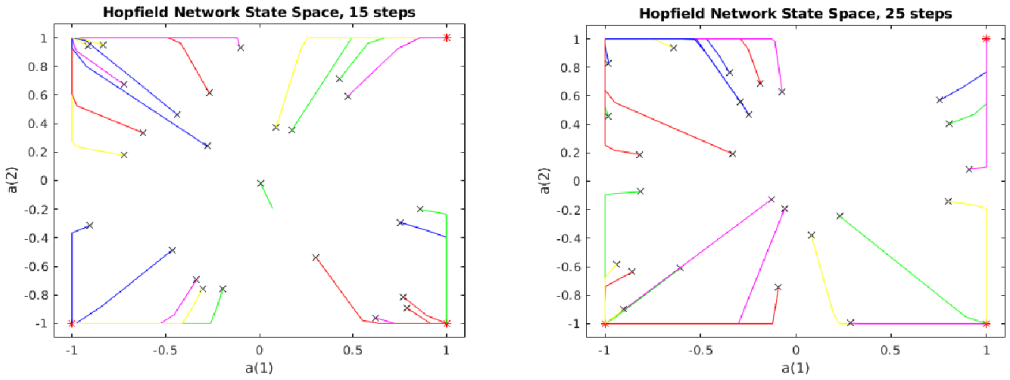


Figure 17: Time evolution in the phase space of 2d Hopfield Model w.r.t. number of iterations required to reach the attractor (right)

For the next task, six symmetrical pairs of points are handpicked as initial points instead of generating random points. The points are [0 0; 0.5 0; 0 0.5; -0.4122 0.6627; -0.9303 0.5157; -0.9142 -0.3143]. A two neuron Hopfield Network is created with three attractors [1 1;−1−1; 1−1] and fifty vectors and is simulated for fifty iterations.

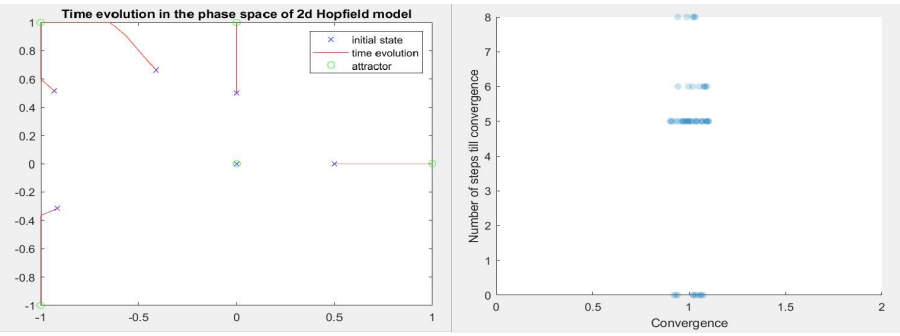


Figure 18: Time evolution in the phase space of 2d Hopfield Model (left) and number of iterations required to reach the attractor (right)

The attracters obtained include the ones stored during the network's creation, but also include two additional attractors. An interesting observation is that one of the attractors is the same as one of the initial points (0,0).

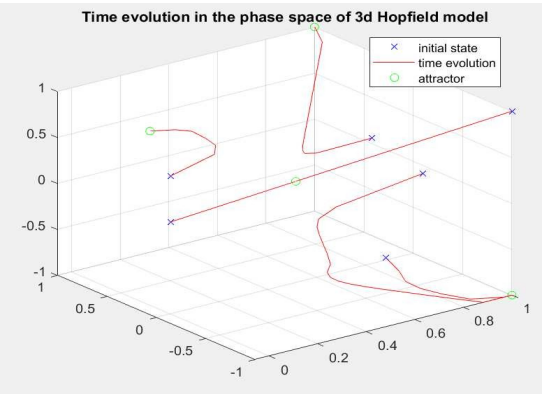


Figure 19: Time Evolution in the phase space of 3D Hopfield model

A similar approach is adopted to create a three neuron Hopfield Network with three attractors [1 1 1; -1 -1 1; 1 -1 - 1]. The network is simulated for fifty timesteps with six symmetrical pairs of points [1, -1, 1; 0 0 0; 0 0 0.5; 0.6627 -0.4122 0.6627; 0.6627 -0.9303 0.5157;0.5157 -0.9142 -0.3143].A similar phenomena can be observed with an additional attractor in the mix along with the original three attractors declared during the creation of the network. For the final task in this section, a Hopfield Network is created with attractors ranging from zero to nine. Next, some noisy digits are added to the network in order to test the ability of the network to retrieve the patterns correctly.

Gaussian noise is added to the digit maps and its level is denoted by a number from zero and ten. A noise level of four is added to the Hopfield network and is made to run for five hundred iterations. We see that for 90 percent of the time our

final results are the same. In the second case noise level is increased to maximum and no of iterations remain the same. We see that there is a big difference in results as compared to first.

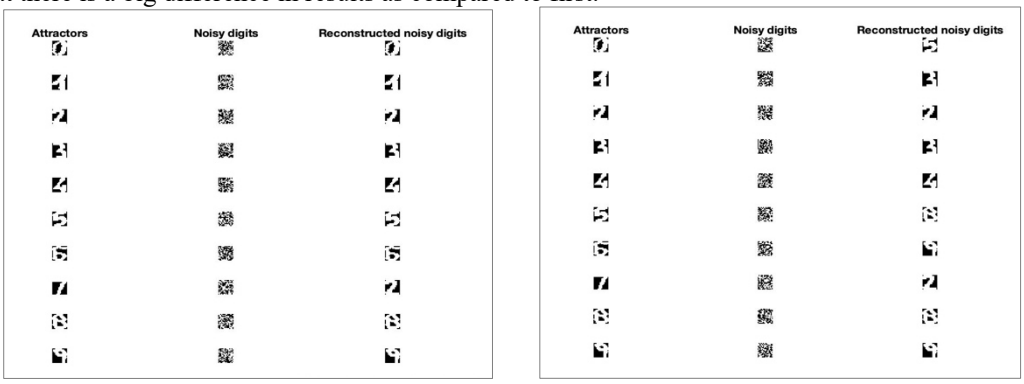


Figure 20: Noise level of five and iterations equal to 500 (left) and Noise level of 10 and iterations equal to 500 (right)

In the second case, noise level are kept at constant and number of iterations are varied. We see that when iterations are reduced to 250, we see only two out of the nine times noisy digits are interpreted correctly. An important thing to consider is higher number of iterations doesn't mean that we can go for as long as possible. There should be a good limit. As I noted that between 500 and 700 iterations maximum accuracy was observed with a noise level equal to five. Beyond that, the accuracy was significantly dropped because of overtraining of data.

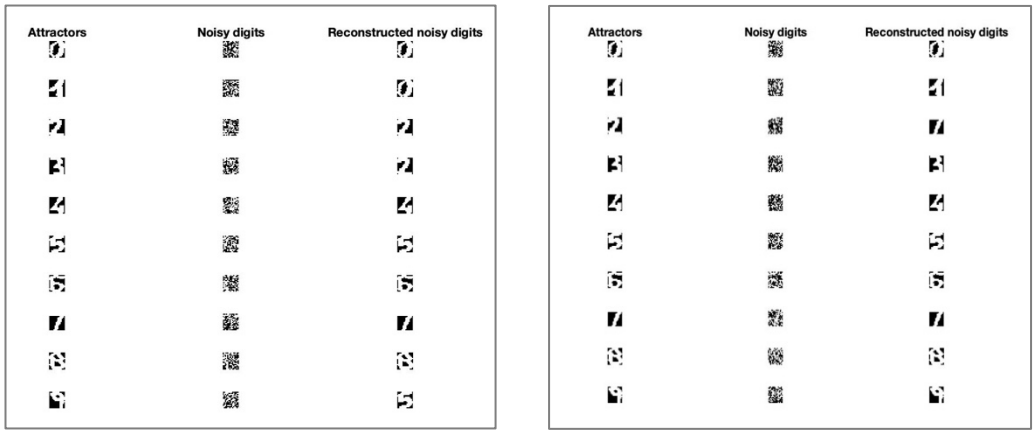


Figure 20: Noise level of five and iterations equal to 250 (left) and Noise level of 5 and iterations equal to 700 (right)

It's evident from the above plot that with least noise and maximum iterations, the success rate of the network is at its peak. The failure count gradually increases with decrease in number of iterations and increase in the level of noise. The performance isn't very dependent on the number of iterations when the noise is zero. Its performance is tolerable when both the noise and the number of iterations is at their maximum. Thus, the Hopfield Network is not very noise tolerant, however, minimal noise with maximal iterations can make the network less prone to overfitting and errors.

2. Long Short-Term Memory Networks

a. Neural Network using Time Series Prediction

In feedforward mode, an MLP with one hidden layer is trained. The Santa Fe dataset, which contains 1000 training data points and 100 test / validation data points, is used. The training and test data are depicted below. The datasets are standardized by calculating the mean and standard deviation. The optimal parameters are then determined using Monte Carlo simulation. The neural network used here is known as a recurrent neural network because it is used iteratively to make predictions on unseen data. It predicts the value at the next timestep using the predicted value from the previous timestep as input.

This technique is based on the time series prediction concept, in which the prediction for a given time is equal to a weighted sum of the previous values up to a certain lag. The tuning parameters are the number of lags and the number of neurons in the hidden layer, and the simulation is repeated five times for each parameter combination because the results tend to fluctuate. [10, 20, 30, 40, 50] and [20, 30, 40, 50, 60] are the lags and hidden sizes used. The function getTimeSeriesTrainData() is used to generate an X&Y matrix for the training and testing phases. The standardized datasets along with the lag pertaining to a particular loop are fed into the function to obtain the time series compatible train and validation datasets.

Because of its versatility and higher reliability factor, the Levenberg-Marquardt learning algorithm is used to train the feedforward neural network rather than the Bayesian Regularization algorithm. The network is trained by varying the lag and layer size. During the training phase, predictions are made on a validation set made up of the training set's last lags (depending on the loop index) at the end of each loop. The standardized data is reduced to its original size and the results are saved. The Mean Square Error and Root Mean Square Error are two comparative descriptors produced by training a network based on each combination of lag and layer size.

MSE	P = 10	P = 20	P = 30	P = 40	P = 50
numN = 20	22.1%	23.9%	2.1%	49.7%	2.6%
numN = 30	9.0%	8.0%	0.8%	2.4%	40.1%
numN = 40	5.6%	5.7%	0.8%	7.6%	27.1%
numN = 50	4.9%	3.1%	0.9%	2.0%	18.1%
numN = 60	6.4%	4.9%	1.5%	30.4%	44.2%

Figure 21: Calculation of Mean square error using default values of no. of neurons and lag

The parameters that appear as optimal in this case are thirty and forty for the number of neurons in the hidden layer and the lag respectively. The optimal parameters are verified visually by applying it on the test set after training a neural network using the LM algorithm using those parameters. The following plots were produced describing the performance.

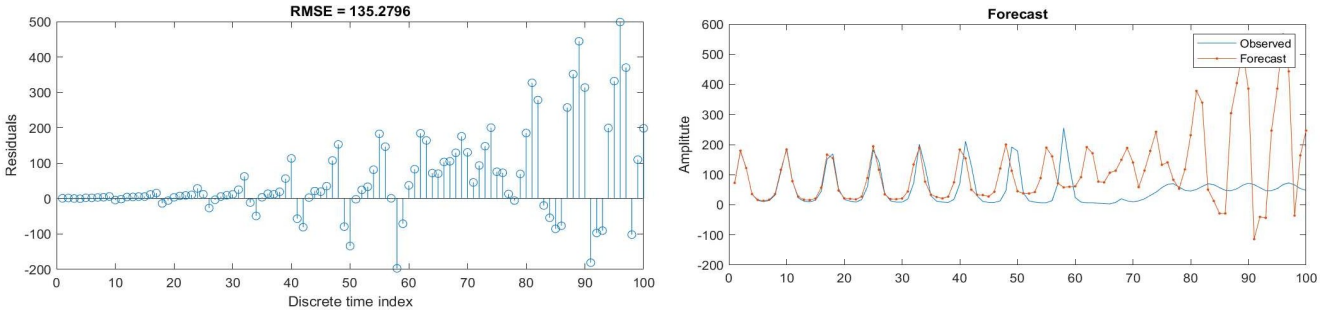
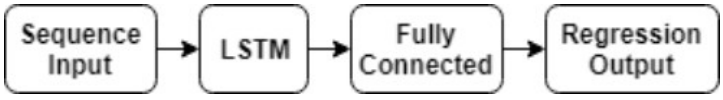


Figure 22: RMSE obtained off the test set (left) and the forecast vs observed curve plot comparison (right)

b. Long Short-Term Memory Network (LSTM)

The advantage of using LSTM over Neural Network is that it allows us to retain past information in the model's training and use it as "memory" to make predictions based on current data and relevant learnings from the past. On each state, LSTM can choose which information to delete from the past, which information to retain, and which to memorize further. This structure effectively addresses the problem of vanishing gradient in Neural Networks, in which the gradient of past moments becomes very small after a number of iterations, preventing the model from retaining past information and informing future predictions. The structure of an LSTM is depicted in the diagram below.



Design process of LSTM

The network is constructed using a time series input layer built out of the Santa Fe dataset used in the previous section, a hidden LSTM layer, a fully connected layer and an output layer. This model architecture helps combat the issue of the vanishing gradient observed often in neural networks which causes the gradients of past predictions to fall rapidly after a number of iterations and to consequently lose past information which hinders its ability to make informed future predictions.

For optimization of the training, the value of lags and the number of neurons is used as the tuning parameters with its possible values being [1, 10, 20, 30, 40, 50, 60, 70, 80, 90] and [10, 20, 30, 40, 50, 60, 70] respectively. Model parameters such as the Initial Learning Rate, Drop Factor List and the Drop Period List are also tuned in the training phase. In order to optimize the model, we test different no of neurons, similar to the Neural Network case, and then try different combinations of Initial Learn rate and Learn Rate Drop Factor. The Learn Rate controls how much the model changes after each iteration with respect the loss gradient. A very small learning rate might lead to long training process as it requires many iterations,

while a large value for this hyperparameter might change the W vector to fast in the training process.

The other 2 hyperparameters that we tune are the Learn Drop Factor and Learn Drop Period. For instance, a drop factor of 0.2 and Learn drop Period of 125, it means that the learning rate gets reduced with a factor of 0.2 every 125 iterations. We train the model, using different values of hyperparameters.

numNlist = [5, 10, 20, 50];

LearnRateList = [0.001, 0.005, 0.01, 0.05, 0.1]; DropFactorList = [0.1, 0.2, 0.5]; DropPeriodList = [25, 50, 125];

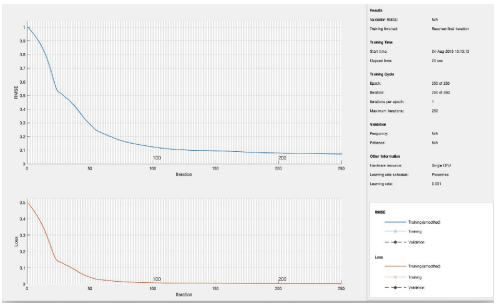
Below the learnings from training:

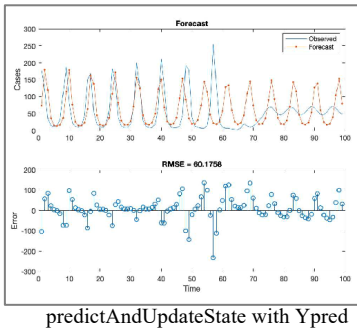
- Small Learn Rates perform better compared to larger values, therefore 0.005 would be our best options. Large values eg. 0.05 or 0.1 result in large errors as the weights are changing too rapidly in each iteration, therefore they are not considered in practice.
- Having a small drop period, i.e reducing the learning rate after 25 or 50 iterations, leads to fast decreasing of the learn rate, therefore the error does not change for the largest part of the training. We set the Drop Period to 125.
- A drop factor of 0.1 reduces the learning rate to 10% of the initial one, while a drop factor of 0.5 only halves it. We notice during training, that a small drop factor of 0.1 or 0.2, especially when initial learning rate is already small, results to inefficient training as there is only minimal improvement of the training error.

The models with the lowest RMSE during training occurred with the below combination of hyperparameters. However, we notice that when we generate the predictions of the model and compare them with the test data, the results are not always satisfactory, as the final RMSE can be high.

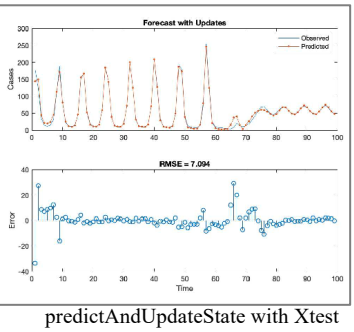
As a second step, we try to predict the test set one step at a time. Moreover, we update the network state at each prediction and we use the previous predictions as input to the function, resulting in a Recurrent Neural Network structure. The results are significantly improved during that step. Below the training process and the predictions of the first and second step for the best performing tuned parameters. The Root Mean Squared Error drops from 60.2 in the first step to 7.1 in the second one.

noN = 20 | Learn Rate = 0.005 | Drop Factor = 0.2 | Drop Period = 125





predictAndUpdateState with Ypred



predictAndUpdateState with Xtest

Which model (RNN or LSTM) do we prefer?

In principle, for a time series prediction, we would choose an LSTM framework over a simple Neural Network. LSTMs use their own predictions to update their state and influence future predictions. They have a special structure that allows them to have memory of past patterns and special operations of forgetting or memorizing new information. Recurrent Neural Networks and especially LSTMs deal effectively with the vanishing gradient problem, that occurs when we need to do backpropagation not only to neurons of the current time, but all neurons of previous times as well, which get closer to zero the more further in the past they are. Due to the above reasons LSTMs have proven to be very effective in various applications that evolve time series predictions, therefore LSTM is my preferred model for this case.

Part III: Deep Feature Learning

1. Principal Component Analysis

We import the dataset and calculate the mean which is equal to 0.2313. Afterwards, we calculate the covariance matrix and its eigenvalues and eigenvectors. From the plot of the diagonal of the eigenvalues, we see that the majority of eigenvalues have a very small value, close to 0. Only a very few eigenvalues have a larger value that grows exponentially. The largest eigenvalues in decreasing order are 21.94, 6.11, 2.76, 2.57, 1.77, 1.58 etc. This means that by reducing the dimensionality as low as 1D or 2D, we will still generate good results.

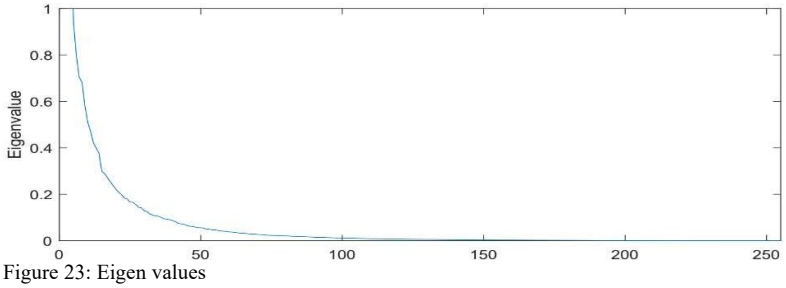


Figure 23: Eigen values

The eigenvalues plot shows that the majority of eigenvalues have a small value. This means that good results can be obtained even after reducing the high dimensions to lower dimensions like one or two! Next, we plot the quality loss against increasing q.

Below the original image of a 3 and the reconstruction images after we applied dimensionality reduction. The more dimensions we use in order to reconstruct the initial dataset, the more details are preserved in the image and the smaller the error we get.

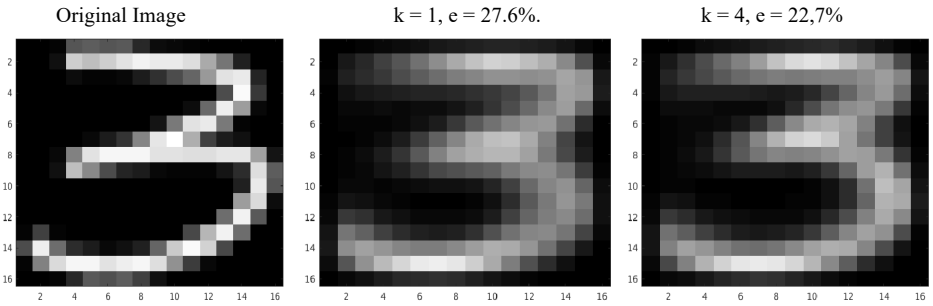
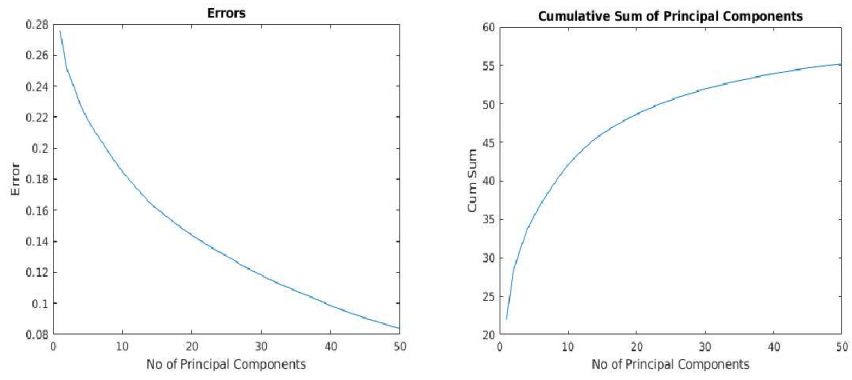


Figure 24: Original image and images with reduced dimensions with varying number of principal components

By plotting the errors corresponding to the No of Principal Components used for the reconstruction process, we see that the bigger the k, the lower the error gets. This was expected as by having more Principal components, we sustain a high-dimensional model, therefore more details over the original input space are maintained.



By setting k in a very large value, eg. k = 256, the error is very low, close to zero. More specifically, the error that we get is 2.9579e-04. Moreover, the reconstructed image that we generate is very close to the original one.

When plotting the Cumulative sum of the first 50 Principal components, we notice that the trend is similar in reverse order compared to the Errors. The sum increased rapidly with the first ks and the trend slows down as the values of the later Principal components become smaller.

Figure 25: Error (left) and Cumulative Sum (right) vs Number of Principal Components. Notice mirror property

2. Stacked Autoencoders

Figure 26: comparison without Fine tuning (left) and with fine tuning (right)

Fine tuning plays a vital role, especially when dealing with autoencoders with multiple layers of encoders. After training just one encoder layer, the softmax produces impressive results as the output of that encoder is already a good enough representation of the input. The effect of fine-tuning becomes obvious when we observe the performance of the model before and after the fine-tuning, with error of 11.1% and 0.3% respectively. The difference is significant. Before the fine-tuning, the NN consists the weights of the Encoder phase of each Autoencoder that were trained individually. Moreover, the Autoencoders in all hidden layers except the last one, were trained with objective to estimate the Input data, therefore they were not optimized. During fine-tuning, the NN gets trained, starting with the weights obtained by the individual autoencoders and then we provide the knowledge of the labeled output values and apply backpropagation.

MLP	With Two Layers	With One Layer
Training Accuracy	99.2	95.8

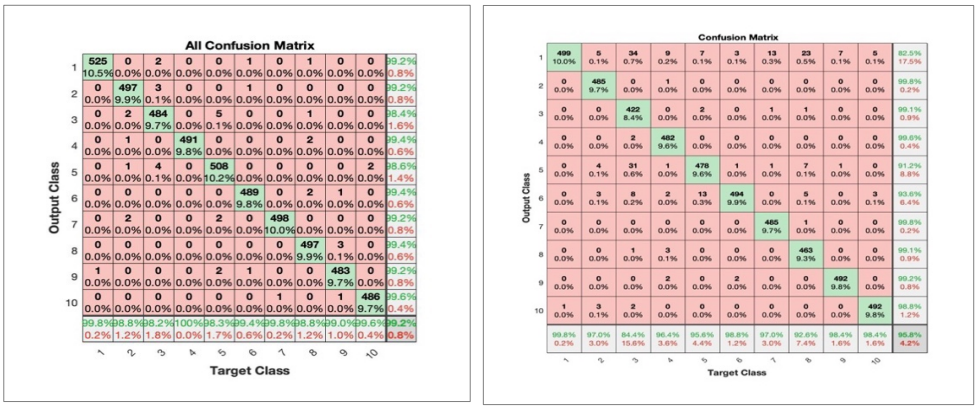


Figure 27: Accuracy comparison of two layer MLP (left) with one layer MLP (right)

The overall performance of the Autoencoder with finetuning is better than the performance of the standard neural network with one of two layers.

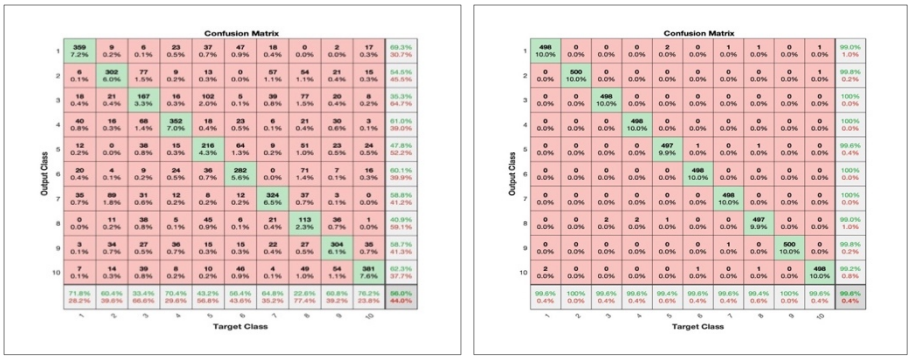
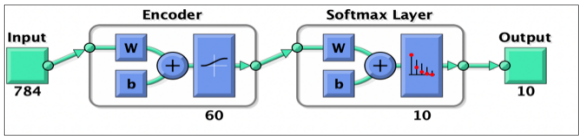


Figure 27: Accuracy comparison of auto encoder (right) with multiple layer MLP (left)

The performance of the autoencoder decrease with the decrease in number of hidden units in the encoder layers. With sixty units in the first encoder layer and with thirty in the second, the test accuracy drops to 56%. However, after applying finetuning, the accuracy shoots up to 99.6, and thus outperforms the standard neural networks with a higher number of hidden units.

With seventy units in the first encoder layer and with ten in the second, a similar phenomenon can be observed where the test accuracy drops to 49.2%. Similar to the previous case, the accuracy shoots up to 99.7% after applying finetuning.



Next, an autoencoder with a single encoder layer is tested against a standard neural network with one or two hidden layers. The autoencoder with just sixty hidden units produce an accuracy 97.42% which shows that even a simple autoencoder with just one layer of sixty hidden units can outperform a standard neural network.

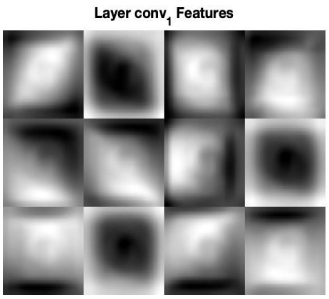
Test Accuracy	Sparsity proportion		Sparsity regularization	
	First layer	Second layer	First Layer	Second Layer
99.6%	0.2	0.2	10	4
99 .1%	0.0	0.4	4	4
92.7% (11 w/o Finetuning)	0.05	0.05	4	4
99.82% (85 w/o Finetuning)	0.3	0.3	4	4
99.82 (85 w/o Finetuning)	0.5	0.5	4	4
99.88 (86 w/o Finetuning)	0.4	0.4	4	4
99.82	0.4	0.4	3	6

It is also evident that using more than two layers of encoders in an autoencoder do perform better than deploying just one layer, however, using just one layer still outperform a multi-perceptron neural network with one or two hidden layers.

3. Convolutional Neural Networks

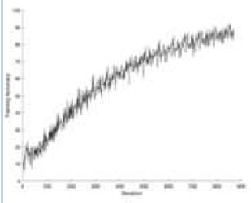
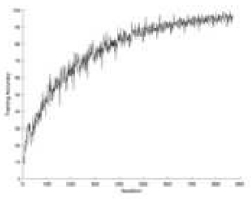
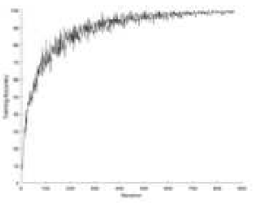
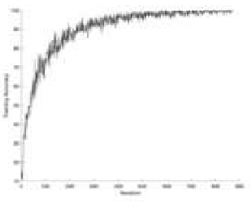
Convolutional neural networks are Deep Learning models that specialize in processing of visual inputs such as images and performing various tasks, for example classification, feature extraction etc.

When running the CNNex.m script, we observe the architecture of the network. The first convolutional layer (layer 2) shows a map of where each feature is located on the initial image inputs. The dimension of the convolutional layer is 96 11x11x3, which corresponds to 96 features/layers and dimension of applied filters of 11x11 with stride 4x4, padding 0 and 3 number of input channels (R B G) that indicates colored images. Stride is the pixel shift of the filter over the input image. Padding is an extra layer added on the border of the image, in order to detect features are the edges of the image as well, that otherwise would have been missed.



The first dimension of the weight matrix is perhaps hardest to visualize. There are 96 individual sets of weights in the first layer. Imagine the 11x11 matrix being replicated 96 times. Each of these 96 things are called channels. Each of these 96 11x11 matrices are initialized with random weights and trained independently during forward / back propagation of the network. More channels learn different aspects of the image and hence give extra power to the convolutional network. But too many channels make it slow to learn and cause overfitting. The features extracted from the first convolutional layer will be used by the deeper layers to learn high level features like nose, eyes, ears, etc.

Indeed, for the second layer, the number of input channels is fifty-five [((227-11)/4)+1], same as number of output channels of layer one. The third and fourth layers are ReLu and Normalisation. Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. We use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization.
The fifth layer is a 3x3 Max Layer Pooling with a stride twin of two, which reduces the input dimensions to twenty-seven [((55-3)/2)+1] as it retains the maximum value only for each 3x3 window. is extremely small in comparison to the 51,529 neurons at the input layer. This is impressive as these thousand neurons describe the key features of the image that will be used for classification.
Question. *What is the dimension of the inputs before the final classification part of the network (i.e. before the fully connected layers)? How does this compare with the initial dimension? Briefly discuss the advantage of CNNs over fully connected networks for image classification.*
A convolutional layer is much more specialized, and efficient, than a fully connected layer. In a fully connected layer, each neuron is connected to every neuron in the previous layer, and each connection has its own weight. This is a totally general-purpose connection pattern and makes no assumptions about the features in the data. It's also very expensive in terms of memory (weights) and computation (connections).
In contrast, in a convolutional layer each neuron is only connected to a few nearby (aka local) neurons in the previous layer, and the same set of weights (and local connection layout) is used for every neuron. This image classification task is a typical use case for convolutional layers as the features are local (e.g. a "nose" consists of a set of nearby pixels, not spread all across the image), and equally likely to occur anywhere (in general case, that nose might be anywhere in the image). The fewer number of connections and weights make convolutional layers relatively cheap (vs full connect) in terms of memory and compute power needed.
We run the CNNDigits.m script trying different architectures. The parameters that I tuned are the filter size and the number of filters per convolutional layer, as well as the number of convolutional layers. Below is the overview of different model architectures, the time of training and accuracy score.

Model	**1**	**2**	**3**	**4**
Architecture	layers = [imageInputLayer([28 28 1]) convolution2dLayer(5,12) reluLayer maxPooling2dLayer(2,'Stride',2) convolution2dLayer(5,24) reluLayer fullyConnectedLayer(10) softmaxLayer classificationLayer()];	layers = [imageInputLayer([28 28 1]) convolution2dLayer(3,12) reluLayer maxPooling2dLayer(2,'Stride',2) convolution2dLayer(3,24) reluLayer fullyConnectedLayer(10) softmaxLayer classificationLayer()];	layers = [imageInputLayer([28 28 1]) convolution2dLayer(3,16) reluLayer maxPooling2dLayer(2,'Stride',2) convolution2dLayer(3,32) reluLayer fullyConnectedLayer(10) softmaxLayer classificationLayer()];	layers = [imageInputLayer([28 28 1]) convolution2dLayer(3,16) reluLayer maxPooling2dLayer(2,'Stride',2) convolution2dLayer(3,32) reluLayer convolution2dLayer(3,48) fullyConnectedLayer(10) softmaxLayer classificationLayer()];
Diagram				
Training	48 sec	40,39 sec	44,18 sec	62,54 sec
Accuracy	0,8344	0,9092	0,9488	0,9668

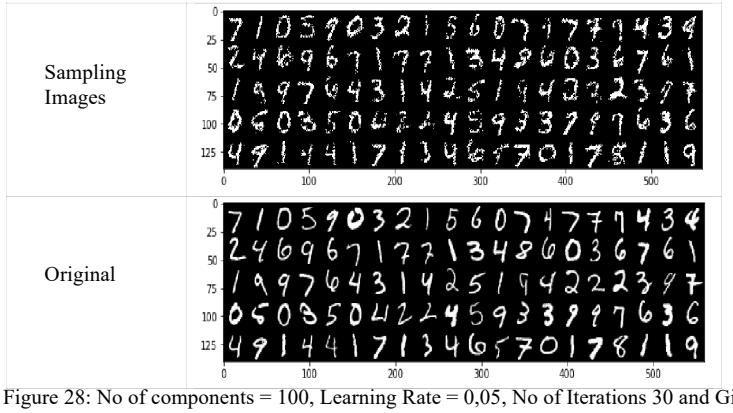
From left to right, we notice that model accuracy increases with the decrease in the filterSize (width and height of filter), from 5x5 to 3x3. Further improvements are also seen when numFilters are increased and additional third convolutional layer is added to the architecture. Applying those changes, the accuracy increases by 13,3 percentage points, from 83,4% to 96,7%, while the mini batch accuracy in the final epochs reaches 100%

Part IV: Generative Models.

1. Restricted Boltzmann Machines

a. Effect of Parameters and Gibbs Sampling Steps

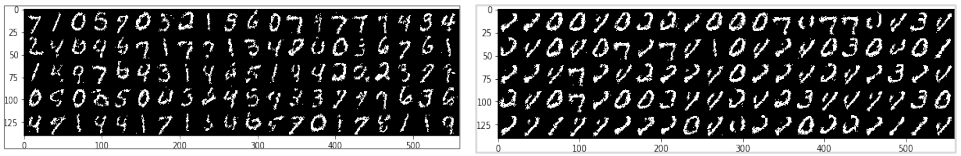
To optimize the RBM model, varying parameters namely, the number of components, the learning rate and the number of epochs/iterations along with changing the number of Gibbs sampling steps, should lead to an optimal combination to produce images close to the original test images (shown below):



The model performs better when increasing the number of hidden units as it learns more patterns from the training set, however the training time is longer. To avoid long training time, we reduce the no of iterations. The learning rate controls how much the weights change after each iteration. A large learning rate results in faster learning, with the downside that it might converge to a sub-optimal set of weights. A small learning rate allows the model to learn a more optimal or even global-optimal set of weights, however the training takes longer as many iterations are required.

Figure 28: No of components = 100, Learning Rate = 0,05, No of Iterations 30 and Gibbs steps = 1, compared to the real images

For the reconstruction of unseen images, 1 Gibbs step is not enough to get a good reconstruction, therefore weincrease this hyperparameter to 10 or 15. The reconstructed images are not perfect, on the contrary they are corrupted.



Increasing the number of components to 150 earns a better result than before, but not with a 1000 Gibbs steps.

Figure 29: 150 components, 0.01 rate, 30 iterations, 5 Gibbs steps (left) and 1000 steps (right)

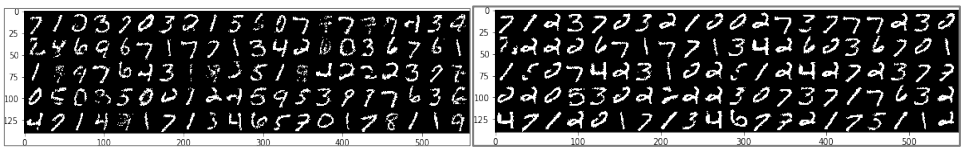


Figure 30: 1000 components, 0.05 rate, 120 iterations, 10 Gibbs steps (left) and 1000 steps (right).

b. RBM to Reconstruct Missing Parts of Image

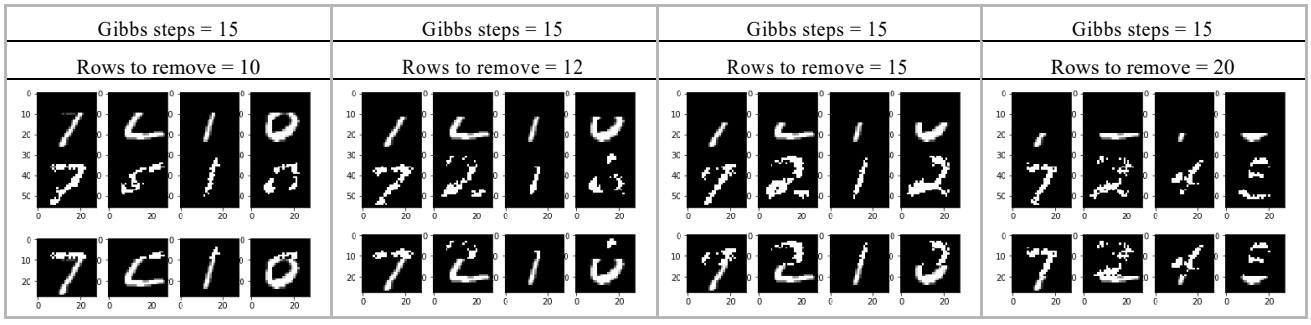
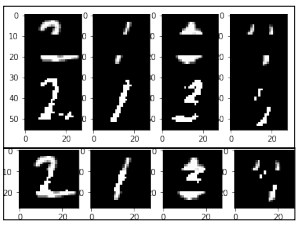


Figure 31: Incomplete images(top) merged reconstruction (bottom)

The more the number of rows missing, the more difficult it gets to reconstruct to the original set of test images. Increasing the number of Gibbs sampling steps helps reconstruct better using more useful features, however, it can only help to a certain extent with more than half of the test image missing.



On removing rows in the middle such as ten to twenty in this particular case, the model attempts to reconstruct the image using ten Gibbs sampling steps. The images with missing middle rows and the reconstructed image is shown. The removed rows are replaced by the reconstructed rows in the merged images on the bottom. The reconstructed images appear corrupted and unrecognizable for the latter half.

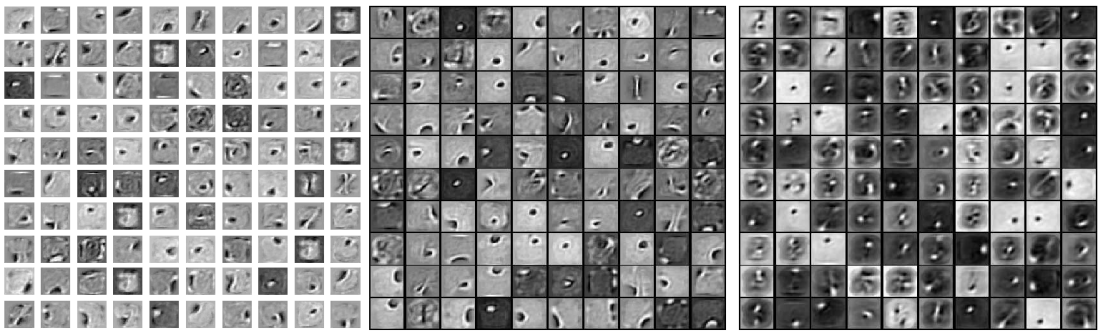
Figure 32: Incomplete images(top) merged reconstruction (bottom)

2. Deep Boltzmann Machines

a. Difference between DBM & RBM

Deep Boltzmann Machines are similar to Restricted Boltzmann Machines, with the difference that they have more hidden layers, stacked one above the other. In a 2-layer DBM, the output of the first hidden layer becomes input for the second hidden layer. The hidden units want to extract features from the visible units and the deeper we go, the more concrete those features are. There is an optimal number of hidden layers for each model that can be determined while tuning.

Figure 33: Features extracted by RBM(left), Features extracted by 1st layer DBM (middle), & 2nd layer DBM (right)



The features from the first layer of the RBM and DBM are similar: they are low level features like edge detection. On the second layer of DBM, the features become more specific like a fragment of an image, a specific shape etc. When adding deeper layers into the model, the features become even more concrete, like a full face in face recognition examples, specific objects or the whole number in the MNIST dataset case. In both the cases, the hidden layer(s) are responsible for extracting features from the visible units. The more the hidden layers are, the more informative & complex the features preserved become, as we go deeper into the model.

b. Sampling & Comparison of New Images of DBM

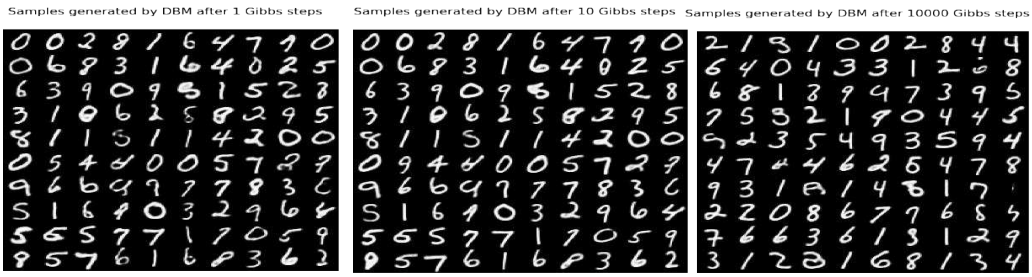


Figure 34: Results generated by DBM after 1 Gibbs steps (left), 10 Gibbs steps (middle), & 10000 Gibbs steps (right)

New images are sampled from DBM. The quality of these samples shows a significant improvement over the samples from RBM (previous section). The samples seem to have more definite structure and the model seems to perform better overall. However, there still exists instances where samples are not recognizable as digits and are deformed. Also, increasing number of Gibbs steps seems to improve the performance, but increasing it too much backfires. DBM manages to outperform RBM due to its multi-layer architecture which helps it to learn more high-level complex features and dependencies, which are then used for reconstruction and generating new samples.

3. Generative adversarial networks (GANs)

a. Deep convolutional generative adversarial network (DCGAN)

GAN is a very interesting concept. The Generator and Discriminator get trained simultaneously by an adversial process. Generator learns to generate images of a class that look "real", while the Discriminator learns to differentiate between real and fake images of that particular class. Stability of GANs is an issue that DCGAN tries to attend to. It uses convolutional stride and transposed convolution for the spatial down-sampling and the spatial up-sampling. The generator is composed of convolutional- transpose layers, batch norm layers, and ReLU activations. The input is a latent vector that is drawn from a standard normal distribution and the output is a RGB image. The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is an image and the output is a scalar probability of whether the input is from the real data distribution. The strided convolutional-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. For a more stable training, the following is proposed by Radford et al:
Strided Convolutions replaces max pooling. Generator and Discriminator learn their own spatial downsampling and

upsampling respectively.
All Fully Connected Layers for all hidden layers are removed in both the models. The last convolution layer of the discriminator is flattened and then fed into a single sigmoid output.
Batch Normalisation normalizes the input to each unit to have zero mean and unit variance which increases the stability. It is applied to all layers except for generator's output layer and the discriminator's input layer (applying to all results in sample oscillation and model instability).
The generator uses the ReLU activation in all its layers except the output layer (Tanh). The discriminator uses *LeakyReLU* for all its layers.

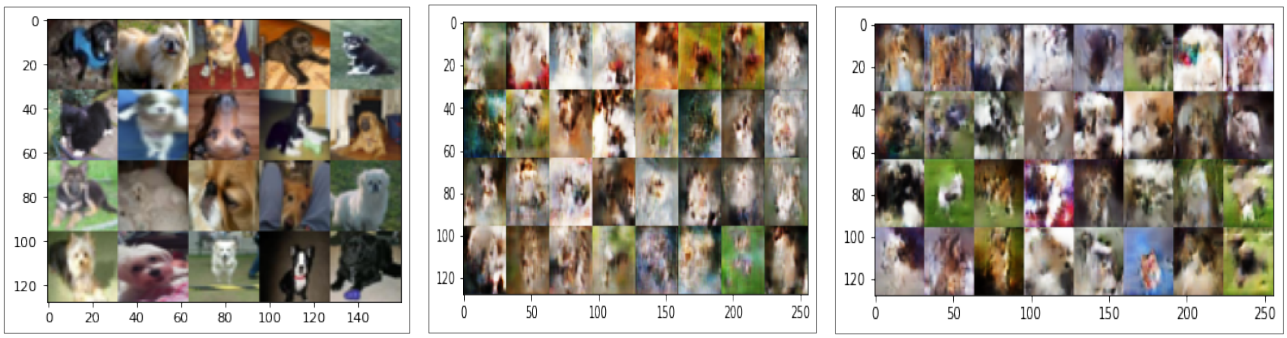


Figure 35: Class 5 real images(left), after 10000 batches (middle), & at 20000 batches(right)

After training the GAN for 20.000 batches we generate the above results. The images resemble dogs, however the images remain blurry. When training is initiated, the Discriminator accuracy is high and Generator loss is also very high. During the training, the Generator improves therefore the error gets smaller and that has an impact on the accuracy of the Discriminator, which gets smaller. The loss and accuracy of the Discriminator and Generator are balancing out, as both models improve in parallel.

During training, the generator tries to outsmart the discriminator by generating images that look more and more like the training images. The discriminator tries to be better at identifying fake images generated by the generator from a set of images containing the real training images and the fake images. The optimal performance is obtained when the equilibrium is reached. The equilibrium is reached when the generator generates images that seem to have directly been sourced from the training data (with best accuracy of 45%, not particularly impressive), while the discriminator's confidence in telling them apart lies at 50%. Increasing the batch size to 50000 will significantly increase the quality of final images but at an expense of extra training time.

4. Optimal Transport

a. Colour Transfer

Optimal transport is used to transfer colour between two images. It maps a pair of meaningful colour palettes and realizes the mapping by transporting the colour histograms onto one another. Colour transfer using optimal transport outperforms other ways of colour swapping techniques (like pixel swapping), by the virtue of its ability of preserving structures while transferring colour.

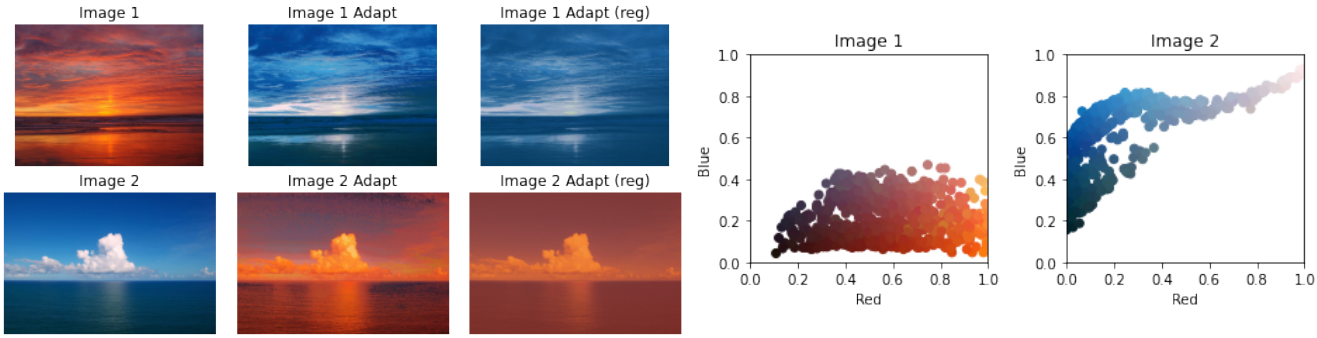


Figure 36: Pair of input images (left), OT using EMD transport (middle), OT using Sinkhorn transport (right)

b. Minimax GAN vs Wasserstein GAN

Wasserstein GAN provides a better way of training the generator model to approximate the distribution of the data from the training set. WGAN replaces the discriminator model with a critic that scores the realness or fakeness of a given image. The training phase of WGAN is more stable, less sensitive to model architecture and selection of hyperparameter values, and causes less mode collapse. Also, the meaningful loss (Wasserstein loss for both models) of the discriminator appears to relate to the quality of images created by the generator. The Wasserstein distance measures a distance of a straight line for transforming one distribution to the other and guarantees continuity and differentiability. WGAN uses either gradient weight clipping or norm penalizing of the gradient of the critic. Below we train a standard GAN and a Wasserstein GAN. Both methods produce bad results due to the simple fully-connected architecture for training efficiency. The Wasserstein GAN tackles better the noise compared to the standard GAN.

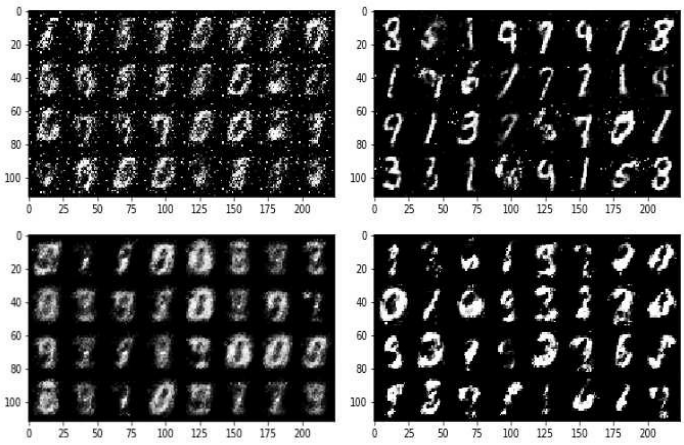


Figure 11 (Clockwise): GAN (5001 Batches), GAN (46001 Batches), WGAN (46001 Batches), WGAN (5001 Batches)