Katholieke
Universiteit
Leuven

MASTER in
ARTIFICIAL INTELLIGENCE

ARTIFICIAL NEURAL NETWORK & DEEP LEARNING

REPORT

Sarah Binta Alam SHOILEE (r0729856)

Academic year 2018–2019

Contents

1 Exercise Session 1

Neural networks is said to be the best performing machine learning approach. To maximize its performance even further, several optimizing techniques have been developed. The best interest of developing those algorithms was to make it computationally less expensive in term of time and speed. But, these algorithm contains it's advantage and disadvantage depending on the applications and architecture-in what model it is being applied. In this report, I am going to illustrates this fact for some given non-linear function and tries to find out what performs better and why.

1.1 Exercise 2

1.1.1 $\mathbf{y = \sin(x^2)}$ without noise

For the function, we try both gradient descent(gd) and gradient descent with adaptive learning rate(gda), they both performs poorly. Up to 1000 epoch they don't converges to minima. The mean square error of their learned function is 0.3479 and 0.1146 respectively(for 1000 epoch). However, the result suggested that "gda" performs better than "gd" in terms accuracy. We tried more epoch, gda converges at 9880 epoch and its not the case for gd. So, we can conclude that "gda" converges faster than "gd".

In Levenberg-Marquardt algorithm, it generates a good result(with sme= .0219) only in 13 epoch to provide the perfect correlation between the input and output. But, it keep convergencing until 1108 epochs for very tiny amount of improvement.

The Quasi Newton algorithm takes around 30-40 epochs to produce a workable solution that manage to draw a correlation between input and output(0.996-0.999). But it takes more time to perform this 30 epochs, around 3 times more on average than the gradient descent approach.

For Fletcher-Reeves conjugate gradient(cgf) algorithm, we need almost double time than gd to compute 35 epoch but cfs's performance is almost converged, except some fl actuation in square mean error after one decimal point. Polak-Ribiere conjugate gradient algorithm(cgp) provides a slightly better result than cgf with same number of epoch.

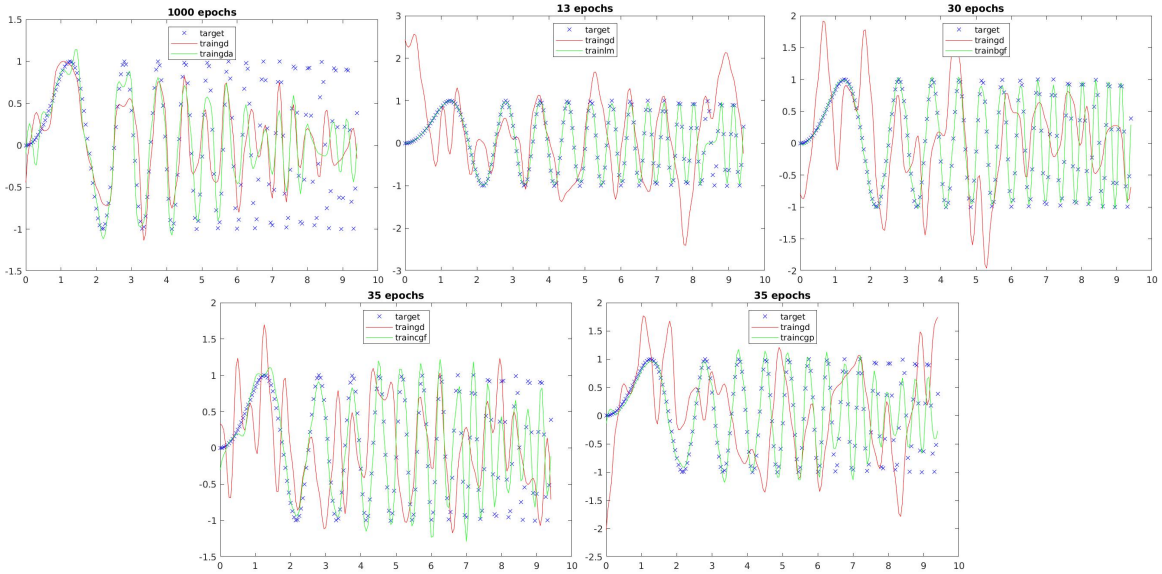

Figure 1: Performance comparison of gradient descent with gradient descent with (a)adaptive learning rate, (b) Levenberg-Marquardt algorithm, (c)quasi Newton algorithm (quasi Newton), (d) Fletcher-Reeves conjugate gradient algorithm and (e) Polak-Ribiere conjugate gradient algorithm

1.1.2 $y = \sin(x^2)$: noisy data

To prepare the data set, the interval of input is used "0.005" which provides 1885 data point. Later, some data points(15%) was chosen and replaced with random value which we can consider as noise. To For noisy data, Levenberg-Marquardt algorithm performs best even after single epoch and successfully draw around reasonable correlation with original data set where gradient descent fails to draw any correlation even after 1000 epochs. We achieve best result for gradient descent with adaptive learning rate with 28 epochs, then it started to be over fitted with the noise of the data set. The same happens to bfg and lm algorithm, after reaching the cut off point it starts decreasing its performance with more epochs. For both noisy and pure data Quasi Newton and Levenberg-Marquardt takes long to converge to a stable point, with 1000 epochs bfg performs better than lm in noisy environment. The conjugate Conjugate gradient converges faster than any other optimization algorithm and the behave more generalized than other algorithm.

1.1.3 Personal Regression

For this task, the data is built from two given input vector(X1 and X2). As per instruction, I build the output vector(Tnew) from these two input vector and also from 5 other vectors vectors(T1, T2, T3, T4 T5) using the following:

$$Tnew = (9T1 + 8T2 + 7T3 + 6T4 + 5T5)/(9 + 8 + 7 + 6 + 5) \tag{1}$$

The constants used in the equation is the largest 5 digits of my student id used in decreasing order. After constructing Tnew, I have collected three independent data sets each with 1000 points with X1, X2 and their corresponding Tnew(from all 13 600 data-points). To make the three set independent from one another, 1000 points for each data set has chosen at random which means the data points chosen from one data set may or may not be present in another. The purpose of choosing three data set is to use them for training, validation and testing of our feed-forward neural network. After training the network with the training set, validation set is used to measure its performance in different settings(number of Nodes, number of Layers and algorithm in use). This help to find the best setting of our network and to avoid over fit. Finally, we test our network with test set and report its performance.

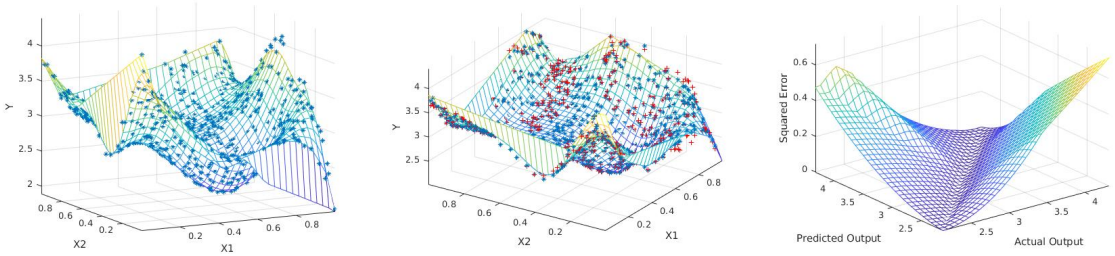

Figure 2: Data representation of (a)training set in 3D space; (b) test set in 3D space-blue '*' are actual data points and red '+' are the predicted values; (c) Error curve of the model

To find the best architecture to find a good model, I made a grid search of number of layer(1-4 layers) and number of neurons(10-50 with interval 10) in each layer for all the optimization algorithm. I always keep the transfer function to "tan-sigmoid", because here we are interested to find the model of data and make decision according to that. It turns out Levenberg-Marquardt algorithm performs best in term of accuracy with 3 layers and only with 10 neurons. Not surprisingly, Levenberg-Marquardt gives better result due to the fact of having less approximation while updating the waits, but also converges faster due to the Hessian calculation. Though it is quite expensive in term of time but we can avoid that by limiting the number of iteration. With only 30 iteration it provides a square means error of 7.0188e-05 on the validation data set.

The root mean square error for the test set is 0.0297 which is much less in comparison to what we got for the validation set. We can improve the result by increasing more data point. We could have achieve better result by changing the learning rate or taking the optimal initial weights, instead of random initial weight. Another approach, could be using the cross-validation method while choosing the best architecture.

1.2 Exercise 5

After trying "trainbr" both noisy data and without noise for y = $\sin x^2$, we came to a conclusion that it performs the same for both but better than all other optimization algorithm except "trainlm". The interesting fact comes up while trying with different number of neurons is that for less number of neurons Bayesian tends to perform better than the Levenberg-Marquardt algorithm. While trying with 20 nodes with 1000 iteration "trainbr" manages to find better correlation. From the results, we can conclude that with less number of neurons "trainbr" performs best for our given dataset.

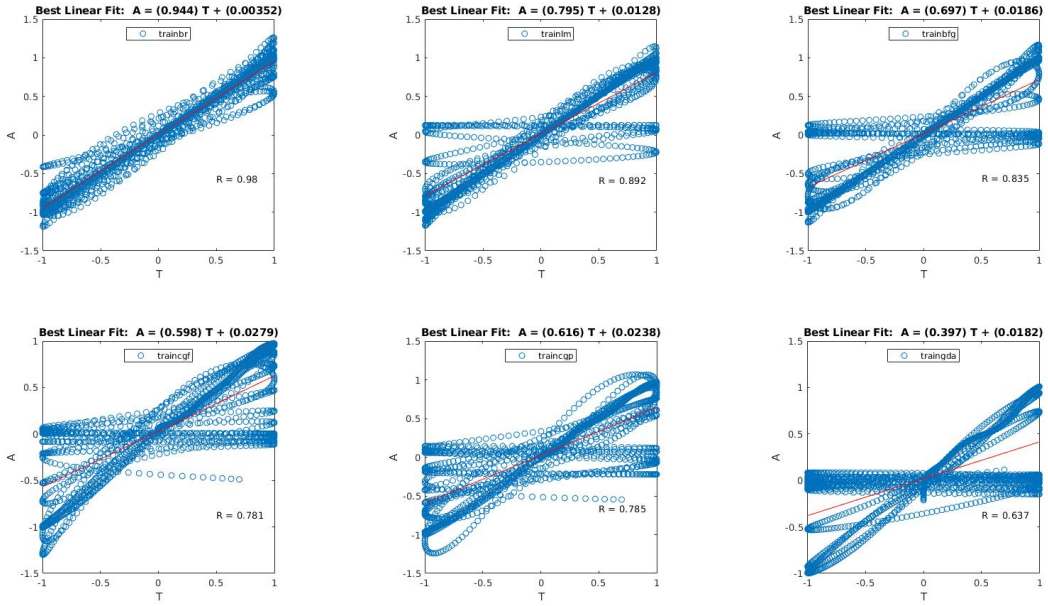

Figure 3: Comparison of "trainbr" with other optimization algorithm with function y = $\sin x^2$(without noise) for 1 layer and 20 neuron architecture(1000 epochs)

2 Exercise Session 2

2.1 Section 01: Hopfield Network

2.1.1 2-Neuron Hopfield Network

For this task, we create a network with three stable points and two neurons. From our observation for 40 iteration, the random initial points always converges to stable equilibrium(attractors), but they also finds an spurious stable points that we didn't store intentionally. We can conclude that the number of real attractor is actually bigger than the number of attractors used to create the network. When we experimented with taking some highly symmetrical points and end up finding some unstable points.

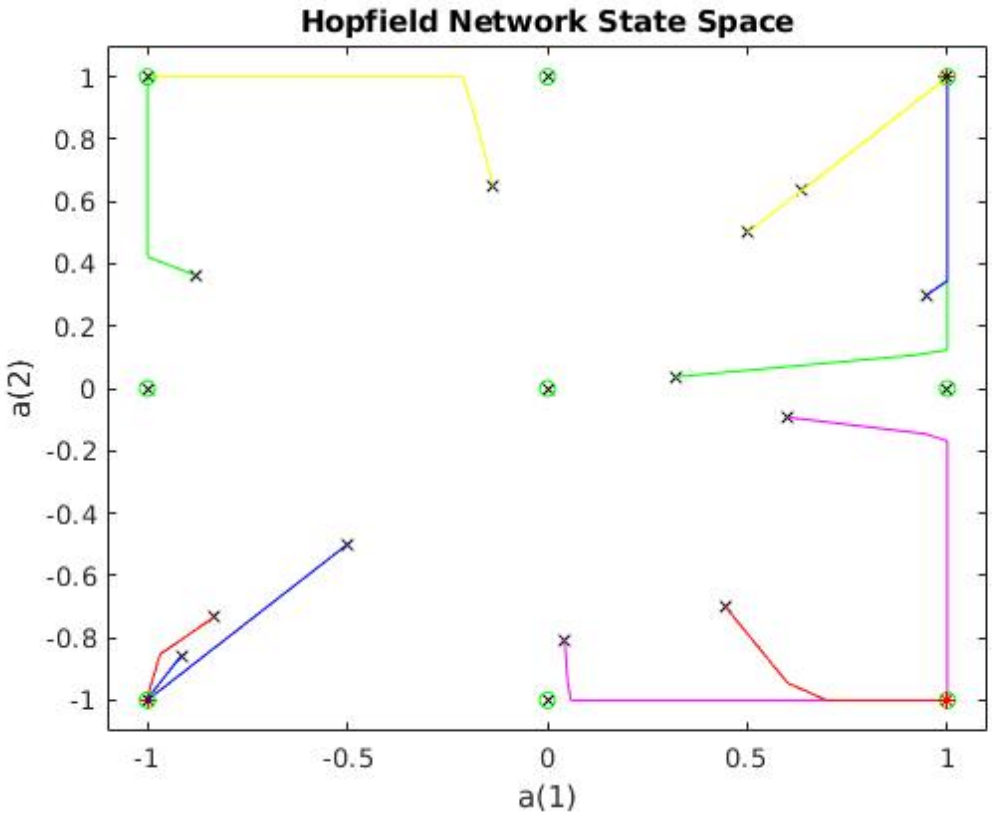

Figure 4: A hopfield network with state space. The red "*" represents the stored stable points and "x" represents the initial points. The green "o" shows the points where any of our initial points converged.

2.1.2 3-Neuron Hopfield Network

We also run the same test for a network with three stable points with three neuron and not surprisingly we get less undesired point than previous. Running the test several time for both with random initial points and intentionally with high symmetrical points, I just encountered with one undesired point (0.36, -0.36, 0.36).

2.1.3 Hopfield Network for hand-written digits

The best use of hopfield network is in pattern recognition with a little noise. For this experiment, we tried to build a hopfield network with ten stable points. Each stable point is a picture of each handwritten digit(0-9). Next, we added some noise(0-1 with 0.1 interval) and tried to recognize its digit pattern and every time we could do that even with 1 iteration. Meaning that the network always converges to stable point irrespective of noise in the image at the initial stage.We further increased the noise even more and notice that though with 10 iteration it converges to some stored image but not always to the desired one due to high amount of noise.

2.2 Section 02

2.2.1 Time-series Prediction

For this experiment, we used a recurrent neural network to predict temperature on the Santa Fe data set. Here, we are interested to predict 100 data points based on 1000 observed data points. We are going to take p(80 in our case) data points to train our model to predict the next data point and likewise keep doing that until we reach to 1000 data point. Using our trained model, we are going to predict the next 100 data points.

We trained our model by only using one layer and with various number of neuron(10-50). We also tried to find out our optimum lag window to produce least root mean square error(rmse) for our prediction with respect to the observed data for these 100 data points. We got our least rsme in a set-up of 40 neuron and with a lag window of length 80. The rmse average value that I got in the given setting is 28.07 with a correlation of 0.87 between observed and predicted data.

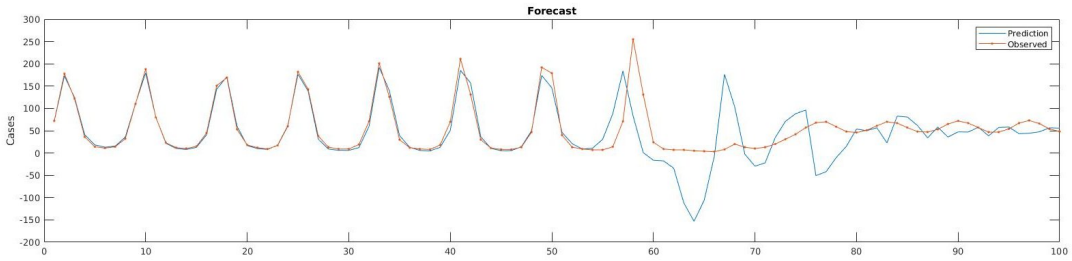

Figure 5: The predicted value using time-series and observed value of 100 days on Santa Fe data set

2.2.2 Long short-term memory network

The long short term(LSTM) is a special kind of neural network which can look back to many time steps and use that information to predict what is going to be next. The recurrent neural network can look back but not very many. LSTM can look back many time steps and make prediction using what it has learned. LSTM has a memory it in to remember information from few steps back. To control when to add item to this memory, pass it to the output or to just forget the information, it maintain its own neural net. Over the time, during the training process, LSTM not only learn information for the prediction network, but also adjust weights for it's input, output and forget gate.

In figure-6 the performance of a LSTM network is presented. This performance is measured on Santa Fe data set where we are provided with temperature data for 1000 days in our training set and also data for next 100 days as test set. The first image in Figure-6 shows the correlation between our predicted data and given data in the test set. This result is generated using 50 Neuron in the hidden layer over 250 iteration during training. In terms of correlation, the prediction is quite good but failed to predict when temperature starts to deviate in a large scale than it's recent days. In the second image of the following figure, we can see the measured root mean square error for the prediction on each day. For the initial days, it performed almost perfect and varied a lot at the end.

The next two pictures in Figure-6 represents LSTM's performance after apply update while training. In the earlier prediction, we used only learned wait and the information stored in the cells but did not update those cell. Now, we are using the cells to contribute on our prediction and also store or using the memory in any other way based on the weights in the gates while new prediction is made. From the bottom two picture of Figure-6, it is clearly evident that using update of cell information improves performance tremendously. While predicting time dependent data like temperature where not only recent data but also data from few days back contribute equally, LSTM always tend to outperforms general recurrent network model.

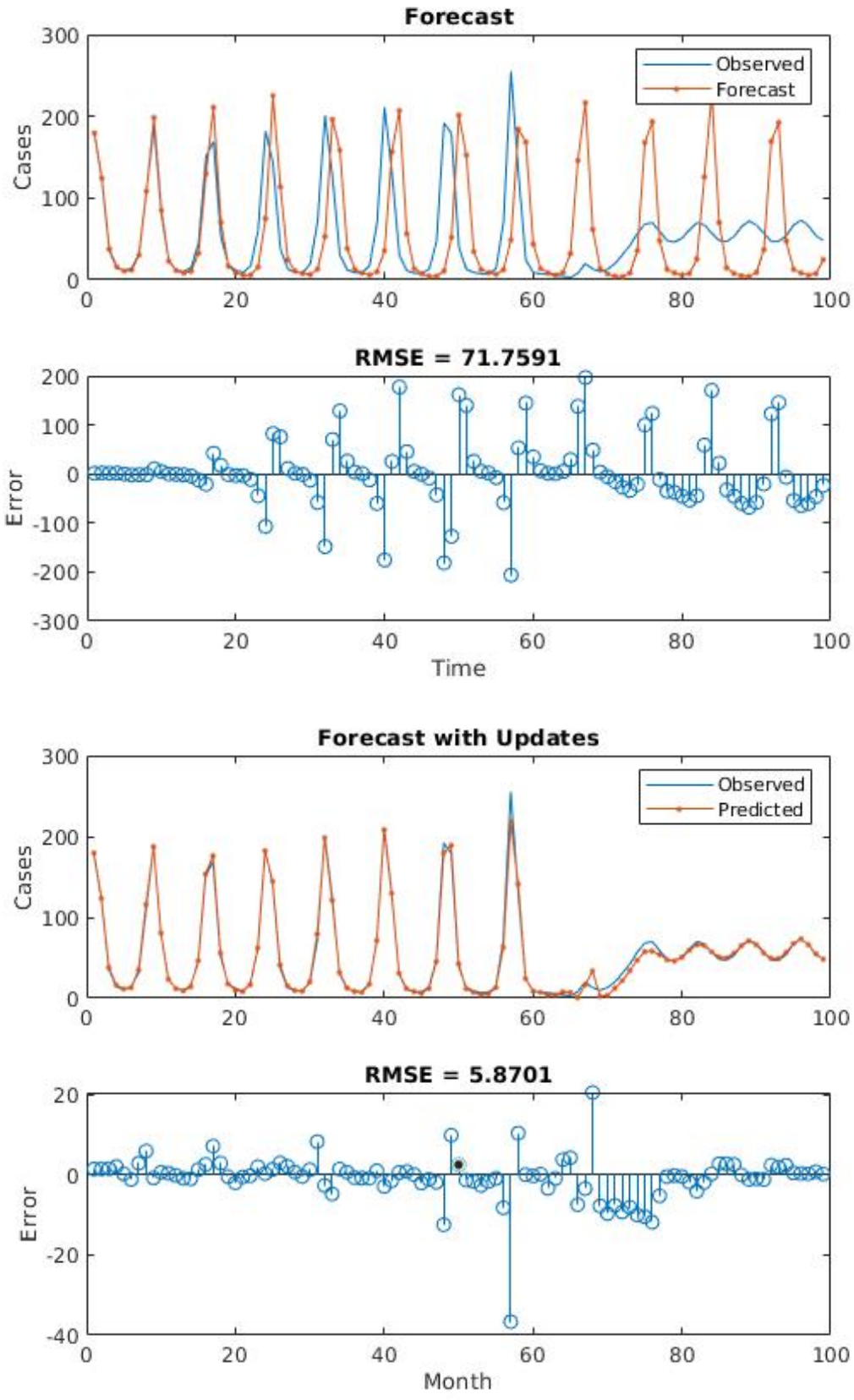

Figure 6: The first two images(from the top) show us the representation of prediction after training model with lstm algorithm. The bottom two picture shows result after update.

3 Exercise Session 3

3.1 Principal Component Analysis on Handwritten Digits

For this exercise, we are given with a data set of 500 images of hand-written digit "three". Each image is of 16x16 pixel(256 points). Figure 7(a) represents a random image from the data set and 7(b) represents the mean image from the overall data set.

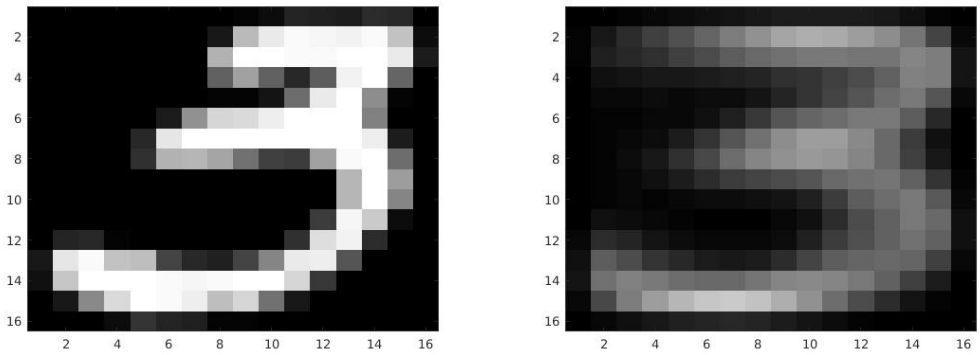

Figure 7: (a)original image in gray-scale and (b)mean of the image data set for hand-written data set three.

To construct PCA of the given data set, we compute the eigen value of the co-variance matrix of the dataset to find the suitable PCA component(k) from its k-largest eigen value. From Figure-8, we can see that eigen value drops larges for the first 50 values and becomes almost zero(0).

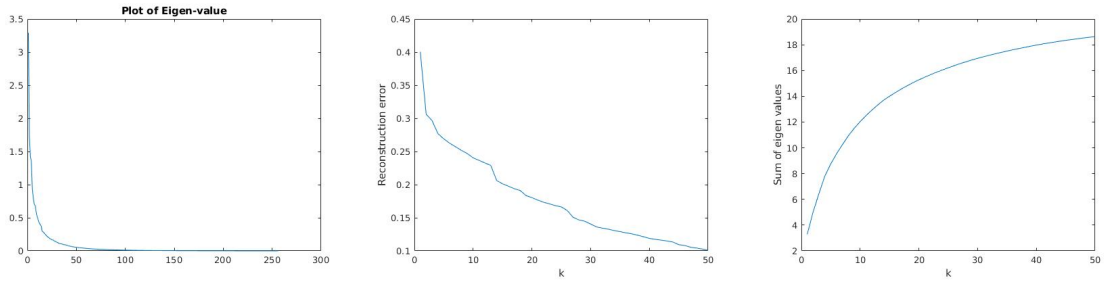

Figure 8: (a)Plot of eigenvalues and (b)squared reconstruction error corresponding to image reconstruction with k=1:50 eigenvalues (c)the sum of eigenvalues w.r.t it's position while ordered as decreasing.

To understand the effect of PCA component, we tried to compress the dimension of image using 1, 2, 3 and 4 PCA component and got the following images after reconstruction(Figure-9). As we can see from the figure, we failed to reconstruct the actual image and the reconstruction error is 0.4, 0.31, 0.30 and 0.28 respectively. When we compress data into lower dimension, we loose lots of information and in turns fail to get back the original image. To check what could be a good PCA component for our data set, we tried all k-components (1:k:256) and calculates the reconstruction error. We take the value 50 as out ideal PCA component as we have only 1% reconstruction error at this point. Figure-8(b) and Figure-8(b) show that the squared reconstruction error induced by not using a certain principal component is proportional to its eigenvalue. Surprisingly, we still get error even if we use PCA component=256 (reconstruction error= 5.86e-16).

3.2 Digit Classification with Stacked Autoencoders and CNN

For this task, we used stacked autoencoder. By trying different parameter(number of layers and numbers of neuron in each layer), we found out that the best result we can get by using two layer with 100 and 50 neurons respectively(with accuracy 99.72%). We can conclude that it is a significant improvement over normal neural network's pattern recognition performance. Using this same architecture for a multilayer neural network, we

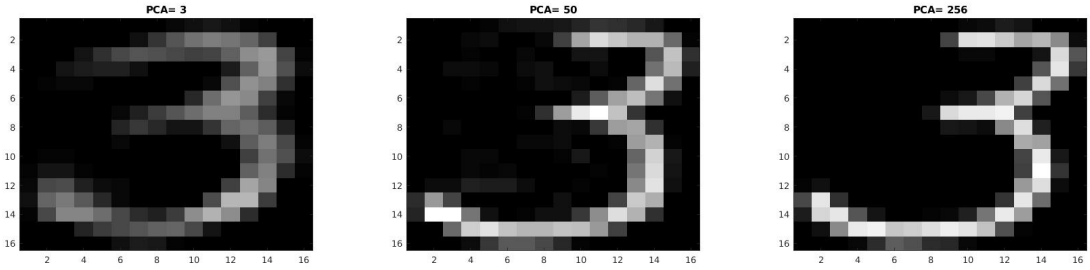

Figure 9: Image reconstruction using PCA component= 3, 50 & 256 respectively.(starting from left)

achieve accuracy of 96.3% on average. The best achieved result for normal neural network is around 97% which we can reach even using one layer with 100 neuron. On the contrary, stacked autoencoder performs better even with one layer than the best performed normal neural network. The point worth mentioning here is that the performance of stacked autoencoder itself is not very promising, but when we go through the step of fine-tuning(weight update with back-propagation), it tremendously improves its performance. This performance describes the mechanism of stack autoencoder quite well, such that when we train our auto-encoder, we do that in an unsupervised manner and tries to learn the feature of the dataset. In the fine tuning phase, the algorithm adjust the weights of its learned features in such a way which maps the output best.

3.3 Convolutional Neural Networks

3.3.1 Observation on "AlexNet"

"AlexNet" is a pre-trained convolutional neural network trained on ImageNet dataset, one of those who gained popularity in image classification task. Altogether, this network has 23 layer which operates convolution and also classification in the deeper layer. The first layer for the convolution uses 96x11x11x3 weights which is represented as 96 kernels(each of size 11x11 for each three channel of the image) After doing the operation of convolution and max pooling, the first the first block of convolutional and pool layer reduce the size of the dimension to 69984 and pass it to the second convolution layer. After applying 5 blocks of convolution and maxpooling, the feature contains 4096 dimension which is used to train a multiclass SVM classifier for 1000 class. The size of the original input dimension has been reduced to 2.64%. Using only a small percentage of it's original dimension CNN manages to represent the features of its original data set for classification.

3.3.2 Digit Classification

For this experiment, we tried convolutional neural network to classify digit(0-9). We tried different value in different parameters and got the result 98.85% at its best. To achieve this result, we design the network with two convolutional layer followed by two Rectified linear transformation. For the first layer, we only used max pooling. For the first convolutional operation, we used 12 kernel of size 5x5 and for the next one we take 24 neuron of the same size. For the pooling operation, we compute the operation on a 2x2 window with stride 2. While observing the network performance in different settings, we found out that increasing the number of layer do not necessarily increase the accuracy. In the same way, if we increase the kernel size and pooling window from our current setting, we get more error. On the other hand, if we downsize the kernel, then it takes more time to produce same performance. Choosing the value of these hyper-paremeter is a design problem for the application in interest and we conclude that the current network setting performance is resonablely serves its purpose.

4 Exercise Session 4

4.1 Restriced Boltzmann Machines

4.1.1 Exercise:01

Restricted Boltzmaan machine is very analogous to principle component analysis, but the use the probabilistic approach in oppose to deterministic ones. It tries to fit the given observation by generating a probability distribution of its feature representation that describe the data as well possible. This is normally a shallow network with a layer of visible and hidden units in it. It is obvious that the number of reconstruction of the input highly depends on the number of hidden components. With just 10 neuron we can barely see any pattern than a noise, but even with the 784 hidden units, it cannot successfully reconstruct the test images perfectly, though show a lot improvement than previous setting.

The details of the observation for 784 components with a learning rate 0.1 and 10 iteration gives us more under standing. We tries to reconstruct the images, it often confuses between (1, 7) or (2, 5, 0). From the resulting images, we can see it mostly generates digit 0, 1 & 2, may be sometimes 4 or 7, but can never generate 3, 5 & 8. We tried with 100 Gibbs sampling and later gradually moves to 500 Gibbs sampling. The interesting fact here is that with more Gibbs sample, it tends to converge to certain pattern while with less Gibbs(but 100), it was identifying the correct digit.

In the effort of down-sizing the number of components we also tried different number of hidden units. Find out, 300 components with 0.1 learning rate performs quite reasonable and with 100 Gibbs sample, we get the best result in this setting. We tested up to 1000 Gibbs value, but with more iteration it just converges to the most common pattern like (0 or 2). While tuning the parameter for iteration, it not necessarily learning to produce more accuracy, but learned new pattern. For example, with 10 iteration the model learns more the pattern of 1 & 7 but for 20-30 iteration it shifted its learning to pattern of "0".

4.1.2 Exercise:02

Taking the tentative best achieved value for previous experiment(no of component= 300; ;earning rate=0.1 and iteration=30), I tried to find out how many missing values the model can generate to reconstruct the image. Up to 15 rows can be deleted and reconstructed in this setting, but depending on which portion image is deleted the model sometimes confuse 3 with 5. The confusion increases higher with increasing number of Gibbs sample. It takes us just 10 Gibbs to reconstruct the image in the giving settings.

To understand the effect of hidden unit on reconstructing images, we tried different value. For 100 Neuron, it is only possible to reconstruct the image after removing 12 rows(+/- 1), but not more than that. With increasing Gibbs value, the reconstruction becomes more prominent. I takes me 1000 Gibbs sample to properly reconstruct the digit. Surprisingly, the reconstruction is pretty similar even if we use 10 Neuron while training but in this case, I had to go up to 1500 Gibbs sample to construct good image of digits.

4.2 Deep Boltzmann Machines

In deep Boltzmaan machine, we stacked more than one RBM together to achieve better pattern recognition. This model learns feature in each layer and in deeper layer then learn complex structure. For a shallow Boltzmaan machine, they tries to map the whole feature as well as possible in just one layer. We can understand it better by looking the filters in RBM which are quite distorted. If we take a closer look, we can see there are some filters which is allowing everything. In other word, we can say that the filtered features are not that strong. In contrast, if we take a look in the weights for the deep Boltzmaaan machine in the first layer, they are making strong of little bit of the image. In the next layer, they are trying to construct more complex feature. In terms of weights, they are putting more emphasis in smaller partition to make features on those are to be more stronger. See Figure-10.

4.3 Generative Adversarial Networks

Generative Adversarial Networks(GAN) is a model of two neural network competing each other in a zero sum game. The two common concept of in this model is "Generator" and discriminator. The objective function for both of them is the opposite of each other, while one model tries to improve itself while other is winning.

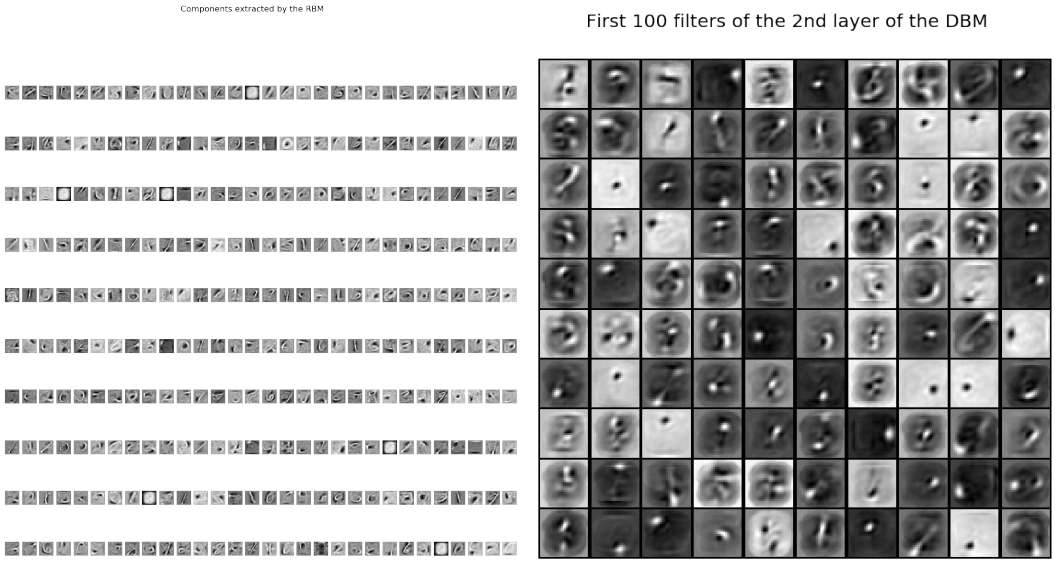

Figure 10: (a)learned weights of RBM and (b) learned weights of DBM.

In this experiment, we applied GAN model on CIFAR data set where we only used image from one class. The interesting fact, I observed here is that with each batch training the loss and accuracy for both Generator and Discriminator is going almost parallel. Meaning that if ones accuracy increases in the next batch training other increases too. This is obvious in this zero sum environment. In such condition, the stability is hard since they can only achieve stability only by producing the value zero. The other problem of stability is the vanishing gradient descent when generator becomes too successful and the gradient descent of generator don't change anymore. But for us it is not the case. We always find the accuracy of the generator is increasing and then again decreasing. This is also true for the discriminator.

Another, theory may describe this instability better which is called "mode collapse". This is a phenomenon while GAN keep generating the same mode rather than others possible modes in data set. When this happens the discriminator tries to improve its performance only targeting this one or two most occurring value and becomes over fitted and in turn drops accuracy.

4.4 Optimal transport

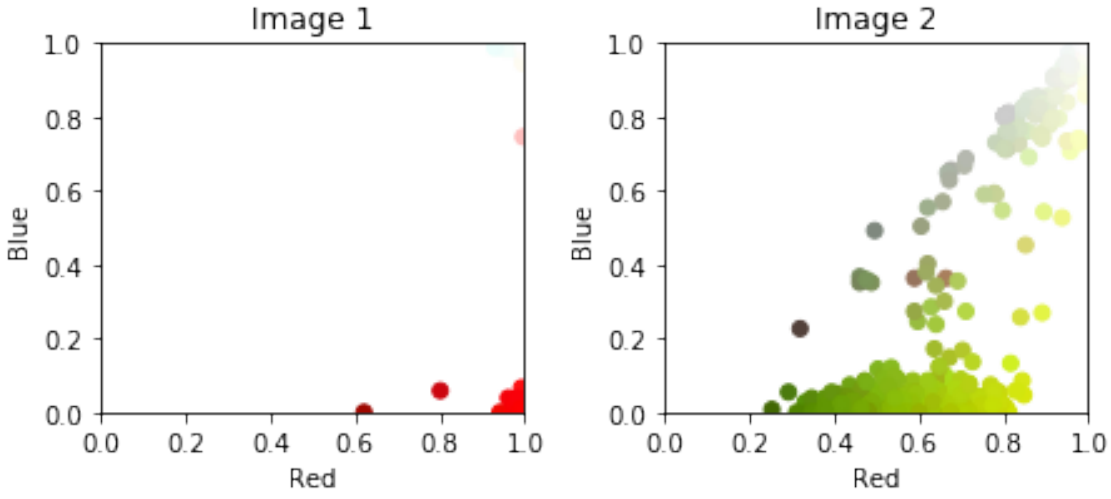

Figure 11: Color probability distribution of two images)

In this given exercise, we used optimal transport to transfer color between two images. The main effort of the optimal transport is to minimize cost function while transferring one state to another. Here, we used transformation in color distribution. In figure 11, we can see the color distribution our first and second image.

In this experiment our interest was to transfer the distribution of red color to the second image and vice verse. To achieve that the algorithm takes every combination between each two image pixels and based on the probability of having certain color and the cost matrix globally determines the Wasserstein distance. Here, we selected two images size of same image. The size of the Wasserstein distance matrix is r*k where r is the number of red pixels and k is the number of green pixels. In Figure 12, we can see the output of the transformation where we successfully swap the pixels between two images based on their intensity.

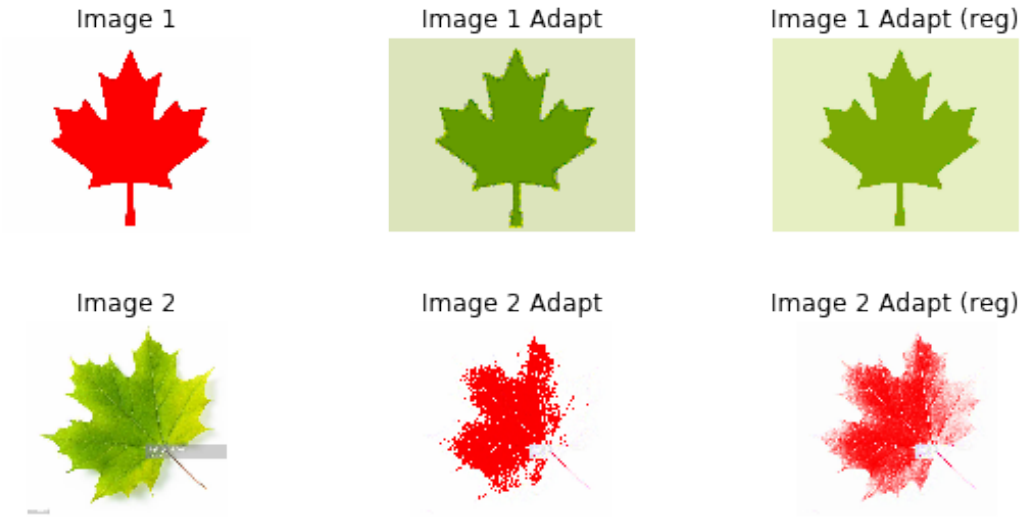

Figure 12: Color transformation using optimal transport

4.5 Wasserstein GAN

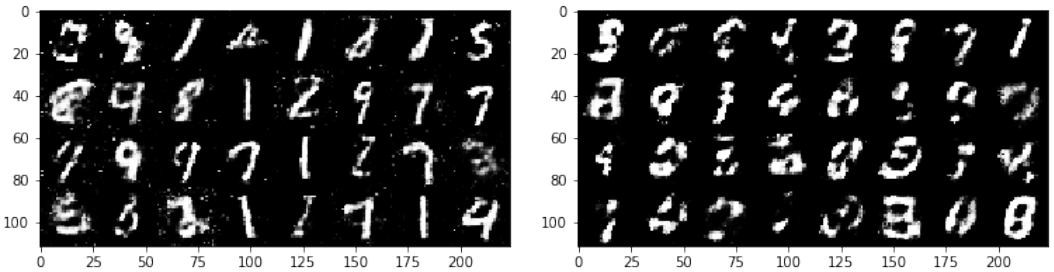

Figure 13: The output of (a)fully connected minimax GAN and (b) Wasserstein GAN from MNIST data set over 100,000 iteration.

The fully connected minimax GAN and Wasserstein GAN is the same mechanism using two different objective function. In fully connected minimax GAN while "Generartor" creates a new data, it tries to match the distribution of new data as close as possible to the distribution of original data by measuring the difference with log-likelihood. Wasserstein GAN uses a difference matrix to measure the error after each data generation. Here, we tried both this GAN approach on MNIST data set. To compare the performance result, configuration used for fully connected minimax GAN and Wasserstein GAN is kept the same for 100,000 iteration (batches = 100000, batch_size=32). The performance is shown in Figure 13. The two model performs quite similar and in terms of generating new pattern minimax GAN did a good job over Wassertein GAN in the given configuration. The interesting fact here is that the right image in Figure-13 generates less noise and more prominent patterns. This effect can be justified by the regularization factor used in measuring Wassetein distance.