Datamining and Neural Network

Zahrasadat Sajjadifar

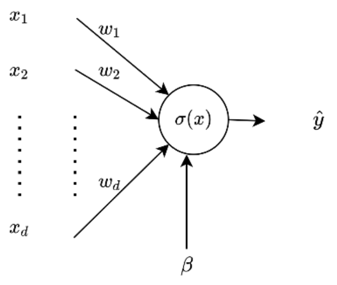
Contents

Assignment 1: Training Algorithms and Generalizations

1.1 THE PECEPTRON AND BEYOND

1.1.1 Perceptron as linear regression

Originally, perceptron consists of one neuron in a single layer and has a *hardlim* transfer function, so it is used as a binary classifier. However, if we apply a linear activation function which is $\sigma(x) = x$ to the dot product of inputs(features) and weights, so we can think of perceptron in this way as a linear regression and for solving linear regression, we need to solve a least square problem, to minimize the square of error between actual values of output and predicted ones.



$$\hat{y} = WX + \beta = \sum_{j=1}^{d} x_j w_j + \beta$$

Figure 1.1 Perceptron as a linear regresion

1.1.2 Function approximation 1

First of all, we need to define the meaning of under fitting and over fitting. Underfitting means that the model does not fit the training set properly, however, overfitting means that the model fits to the training set too much, in a way that it even captures noise and we will not have good prediction for new incoming data in this problem, in other word, the generalization of our model is weak. In this question we have a nonlinear function of $y = -\sin(0.8\pi x)$. A linear model cannot capture the sine wave that we have adequately, and if we fit a linear model to it, we have underfitting problem. In other word, linear model does not have enough degree of freedom to represent this nonlinear function, so we need to increase the order of our model.

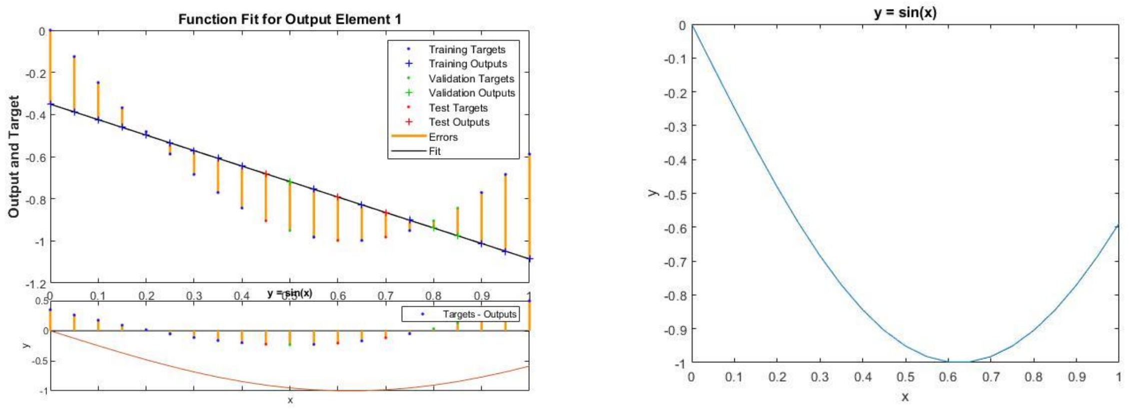


Figure 1.2 $y = -sin(0.8\pi x)$ for function approximation problem. (right), result of training model with one hidden unit and *purelin* (linear activation function). (left)

1.1.3 Function approximation 2

In this question, we train a neural network with one hidden layer consist of two units with *tanh* activation function. This model is a nonlinear model which can be used to approximate the function that we have in previous part. In Figure 1.3, we can find the result.

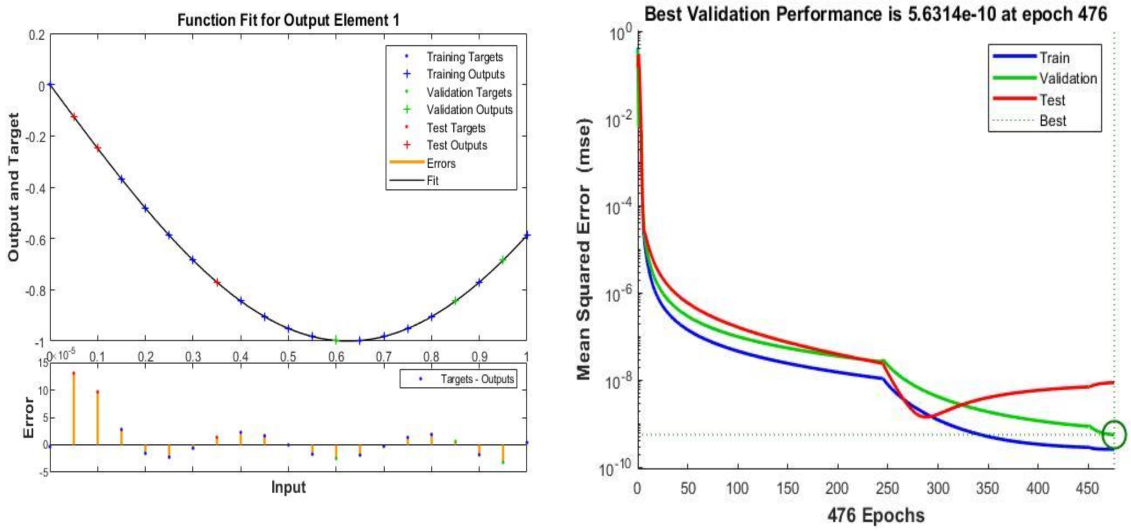
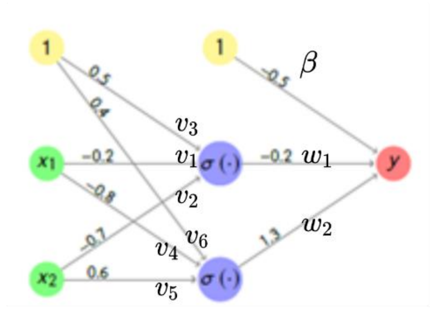


Figure 1.3 result of training network with one layer and two neurons for function approximation.

1.2 BACK PROPAGATION IN FEEDFORWARD MULTI- LAYER NETWOKS

1.2.1 Backpropagation



$$W = [w_1\ w_2] \quad = \quad [-0.2 \quad 1.3\ ]\ , \beta = \ -0.5$$

$$V \ = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 \end{bmatrix} = \begin{bmatrix} -0.2 & -0.7 & 0.5 \\ -0.8 & 0.6 & 0.4 \end{bmatrix}$$

$$X = [x1 \quad x2 \quad 1\ ]^T$$

$$y' = W\big(\sigma(VX)\big) \ + \ \beta$$

$$y' = w_1\, \sigma_1 + w_2\, \sigma_2 + \ \beta = w_1\, \sigma(v_1 x_1 + v_2 x_2 + v_3) + w_2\, \sigma(v_4 x_1 + v_5 x_2 + v_6) + \beta$$

$$data\ points\colon \{(x_1, x_2, y)_i\}_{i=1,2,3} = \{(1,1,1), (2,-1,-1), (3,0,-1)\}$$

$$L(W\ , V\ , \beta) = \sum_{i=1}^{3} \frac{1}{2}\ (y'_i \ - y_i)^2$$

Using chain rule to compute derivatives, and the hint $\frac{\partial \sigma(t)}{\partial t} = \ \sigma(t)(1 - \sigma(t))$. All derivations are a summation over all data point. For simplicity, we just write the main derivation.

$$\frac{\partial L}{\partial w_1} = \ \frac{\partial L}{\partial y'} \ \times \ \frac{\partial y'}{\partial w_1} \ = (y' - y).\, \sigma(v_1 x_1 + v_2 x_2 + v_3)$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial w_2} = (y'-y).\sigma(v_4x_1 + v_5x_2 + v_6)$$

$$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \beta} = (y'-y)$$

$$\frac{\partial L}{\partial v_1} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \sigma_1} \times \frac{\partial \sigma_1}{\partial v_1} = (y'-y).w_1.\sigma(v_1x_1 + v_2x_2 + v_3)[1-\sigma(v_1x_1 + v_2x_2 + v_3)\,]x_1$$

$$\frac{\partial L}{\partial v_2} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \sigma_1} \times \frac{\partial \sigma_1}{\partial v_2} = (y'-y).w_1.\sigma(v_1x_1 + v_2x_2 + v_3)[1-\sigma(v_1x_1 + v_2x_2 + v_3)]x_2$$

$$\frac{\partial L}{\partial v_3} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \sigma_1} \times \frac{\partial \sigma_1}{\partial v_3} = (y'-y).w_1.\sigma(v_1x_1 + v_2x_2 + v_3)[1-\sigma(v_1x_1 + v_2x_2 + v_3)]$$

$$\frac{\partial L}{\partial v_4} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \sigma_2} \times \frac{\partial \sigma_2}{\partial v_4} = (y'-y).w_2.\sigma(v_4x_1 + v_5x_2 + v_6)[1-\sigma(v_4x_1 + v_5x_2 + v_6)\,]x_1$$

$$\frac{\partial L}{\partial v_5} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \sigma_2} \times \frac{\partial \sigma_2}{\partial v_5} = (y'-y).w_2.\sigma(v_4x_1 + v_5x_2 + v_6)[1-\sigma(v_4x_1 + v_5x_2 + v_6)]x_2$$

$$\frac{\partial L}{\partial v_6} = \frac{\partial L}{\partial y'} \times \frac{\partial y'}{\partial \sigma_2} \times \frac{\partial \sigma_2}{\partial v_6} = (y'-y).w_2.\sigma(v_4x_1 + v_5x_2 + v_6)[1-\sigma(v_4x_1 + v_5x_2 + v_6)]$$

$$w_1^{new} = w_1 - \alpha \frac{\partial L}{\partial w_1} \quad , \quad w_2^{new} = w_2 - \alpha \frac{\partial L}{\partial w_2} \quad , \quad \beta^{new} = \beta - \alpha \frac{\partial L}{\partial \beta}$$

$$v_1^{new} = v_1 - \alpha \frac{\partial L}{\partial v_1} \quad , \quad v_2^{new} = v_2 - \alpha \frac{\partial L}{\partial v_2} \quad , \quad v_3^{new} = v_3 - \alpha \frac{\partial L}{\partial v_3}$$

$$v_4^{new} = v_4 - \alpha \frac{\partial L}{\partial v_4} \quad , \quad v_5^{new} = v_5 - \alpha \frac{\partial L}{\partial v_5} \quad , \quad v_6^{new} = v_6 - \alpha \frac{\partial L}{\partial v_6}$$

Now start the gradient descent algorithm for the datapoint that we have, with $\alpha = 1$:

- Feedforward: $X = [1 \;\; 1 \;\; 1] \;\; , \; y = 1 \;\; \rightarrow \; y' = W(\sigma(VX)) + \beta = \;\; 0.1345$
  $X = [2 \;\; -1 \;\; 1]\,, \; y = -1 \; \rightarrow \; y' = W(\sigma(VX)) + \beta = -0.4536$
  $X = [2 \;\; -1 \;\; 1]\,, \; y = -1 \; \rightarrow \; y' = W(\sigma(VX)) + \beta = -0.4400$
- Backpropagation:
  $dW = [0.2957 \quad -0.3316], dV = \begin{bmatrix} -0.0889 & 0.0650 & -0.0097 \\ 0.1237 & -0.3650 & -0.1156 \end{bmatrix}, d\beta = 0.2409$

  $W^{new} = [-0.4957 \;\; 1.6316], V^{new} = \begin{bmatrix} -0.1111 & -0.7650 & 0.5097 \\ -0.9237 & 0.9650 & 0.5156 \end{bmatrix},$
  $\beta^{new} = -0.7409$

1.2.2 Function Approximation

In this part, three different algorithms, *adaptive learning rate*, *BFGS Quasi Newton*, and *Levenberg-Marquardt*, have been compared with *gradient descent* algorithm in terms of performance, speed, generalization and noise, for approximating the function which is $y = \sin(x^2)$, $x \in \left[0, \frac{\pi}{3}\right]$. We have 200 datapoint, and the neural network has 2 hidden layers, each with 10 units (10*10 + 10 + 10 parameters<200, not overparameterized). Number of epochs for each algorithm is 1000, and We put the same weight and bias for each algorithm. These settings are unchanged during the experiment to have more precise comparison. For this comparison, we use time and correlation coefficient R and RMSE and do the experiment for two stage, noiseless data and noisy one. Furthermore, we change the code "algorithms. mlx" in order to implement the networks for 50 iteration and at the end average time and accuracy over all iteration for each algorithm, in order to have more precise result and consider the fact that in training NN, by choosing different initial values, we may end up to different local minima. In Figure 1.4, blue bars, you can find the bar plot of R, time value, and RMSE for different algorithm. Now, we train the same network as previous part, this time with adding gaussian noise to the dataset with standard deviation of 0.3. Different algorithms performance and speed can be compared in Figure 1.4, red bars. As we can see, noise increase the RMSE of all algorithm and we can see high increase in the RMSE of *Levenberg-Marquardt* for noisy data compare to noise less data, we can say it is more sensitive to noise.

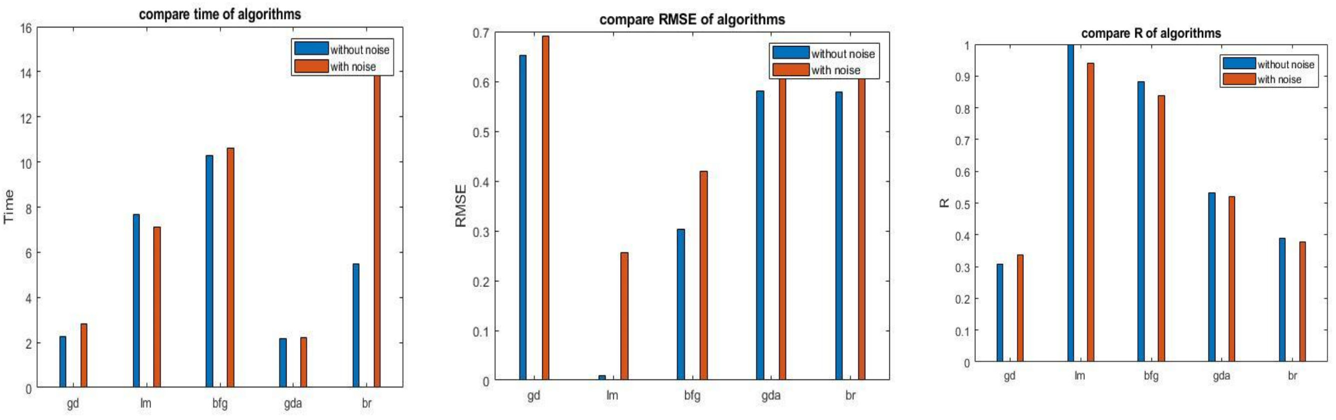


Figure 1.4 RMSE, correlation coefficient, time of training different algorithms

The best performance on our dataset and with our fixed NN is for *Levenberg-Marquardt* which has high accuracy and speed. In general, *gradient descent* has the lowest performance among other algorithms.

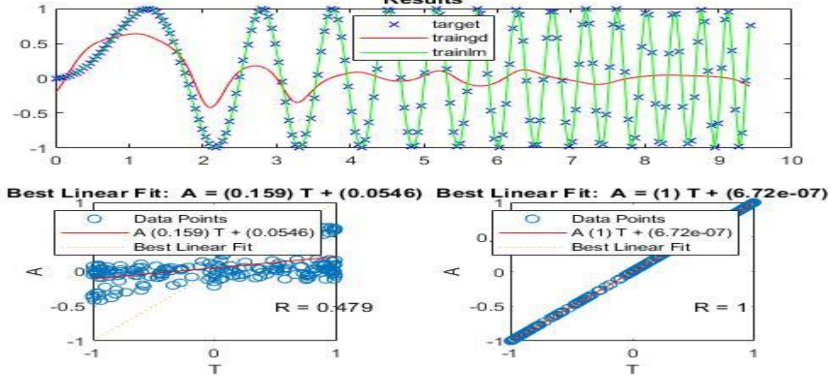


Figure 1.5 Gradient Descent vs Levenberg-Marquardt

1.3 PERSONAL REGRESSION EXAMPLE

Define datasets: In this exercise, we are going to approximate a nonlinear function using NN. The dataset has 13600 data point, two feature(inputs), and one output. To start, we need to choose 3000 number between 0 and 13600 as the indices of our datapoints. To do this, we use MATLAB command *[-,index] = datasample(data,k,'replace',false),* whit *k=3000* and *data=1:13600*, which will return *k* observations sampled uniformly at random, without replacement, from the data in *data*. We cannot use *randi* command because we need to make sure that our choices are not the same (should be without replacement) in order to make sure that we do not have same data in training, validation, and test set. We make our dataset using *X = [X1(index), X2(index)], Y = Tnew(index),* in which *Tnew* is our target values calculated in a way that is explained in question using student number. Now we divide this dataset to train, validation, and test sets each one with 1000 point.

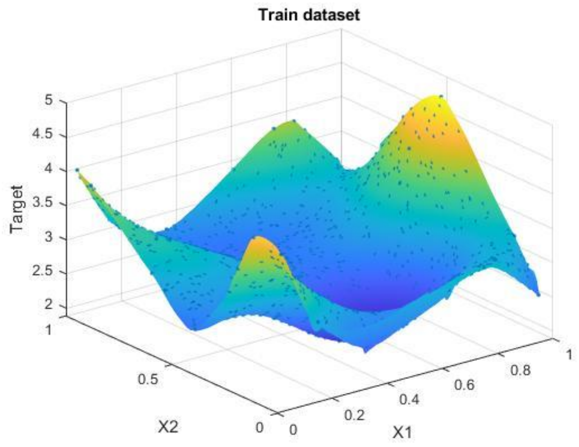


Figure 1.6 Train dataset, dot points are our exact data points, the surface is created from data points using *griddata* command in MATLAB.

Build and train feedforward neural network: we choose an artificial neural network with two layers with the same number of neurons, and with *tanh* activation function for two hidden layers and linear activation function for output layer since we want to approximate the function, and the learning algorithm is *Levenberg-Marquardt* epochs based on the result that we discussed in previous part, it has good performance and speed. We choose 5 different number of neurons in the interval of 10:10:50 and 500 epochs, and for each one run NN for 10 iteration and calculate the RMSE between the predicted and main result on validation set. At the end, average the RMSE over all iterations for each number of neurons to find best possible choice. we can find out from Figure 1.8 that 20 as the number of neurons is a proper choice in our case, since by increasing the number of neurons we overparameterize the network which leads to the overfitting problem and as we can see the RMSE of validation set increases.

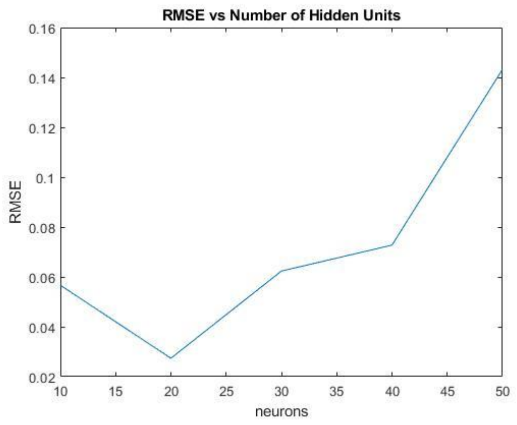


Figure 1.7 RMSE of validation set after 500 epochs for different number of hidden neurons.

Performance assessment: The final network is trained for 1000 epochs, and for reporting the final RMSE, we run the training for 5 iterations again and get the $\boldsymbol{RMSE = 7.56\,e^{-4}}$**.** Figure 1.8 shows the test set function and the one which is predicted by network. We can see that they are almost the same. In order to improve the performance of our network, we try *trainbr* algorithm for training the network. This algorithm, as we will see in next part, is a *Bayesian Regularization* and by adding regularization we limit the norm of the weights not to be increased too much and then avoid overfitting. This algorithm does not use validation set for tunning hyperparameters. The performance with this method is $\boldsymbol{RMSE = 3.91\,e^{-5}}$. We can see the loss curves of train, validation and test set on Figure 1.9 for both algorithms that we tried, and it is clear that *trainbr* outperforms.

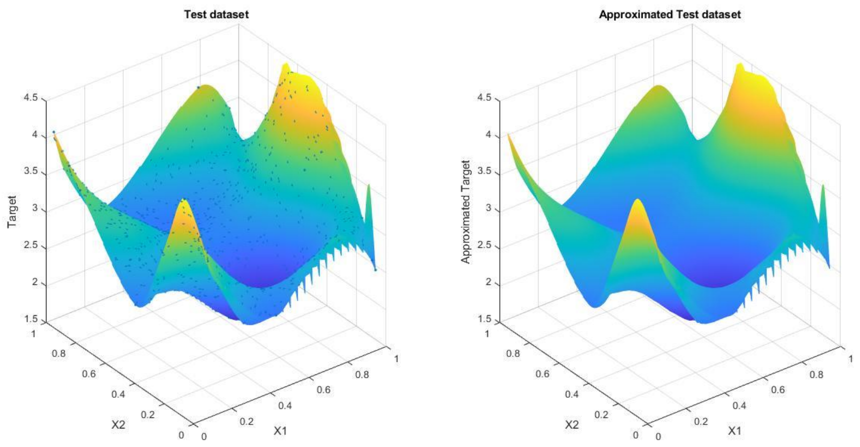


Figure 1.8 The main test dataset and the approximated one using neural network

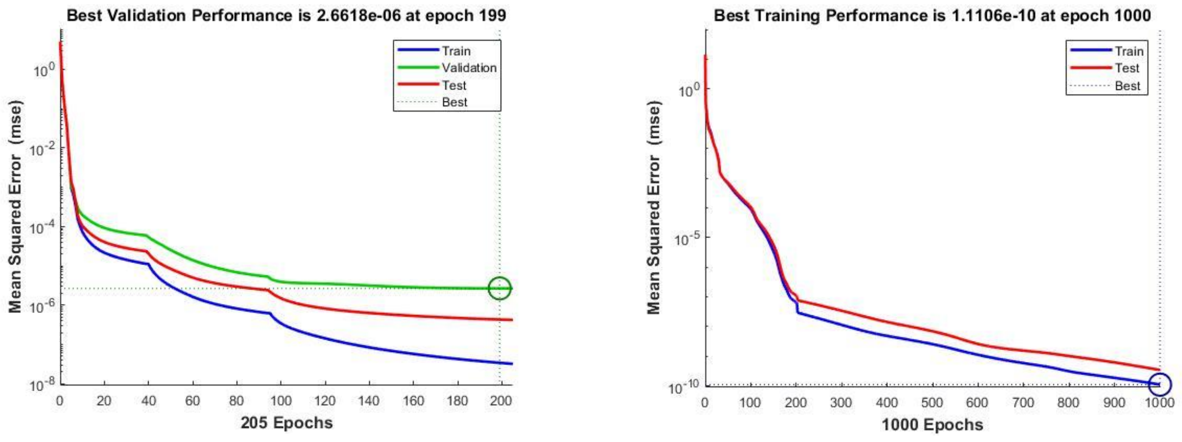


Figure 1.9 Loss curves of *trainlm* algorithm without regularization and *trainbr* algorithms with regularization.

1.4 BAYESIAN INFERENCE

For this part, we do the experiment as we had in 1.2.2 with 10 iterations, and we see that for noiseless and noisy data the *Levenberg-Marquardt* algorithm performs better than *trainbr* and has a much smaller RMSE. Also, we try overparameterized network, larger number of parameters than datapoints. For this, we increase the number of neurons in each layer to 30 (30*30 + 30 + 30 parameters>200, overparameterized), and train the network for 1000 epochs, for noisy and noise less data. For noisy data, still *Levenberg-Marquardt* performs better, but for noise less data, we get a better result and smaller RMSE for *Bayesian Regularization*. The result is $\boldsymbol{RMSE = 2.2\, e^{-5}}$ and $\boldsymbol{RMSE = 9.2\, e^{-5}}$,which can be found in Figure 1.10. So, we can conclude that, for overparameterized network, we can use *Bayesian Regularization* algorithm to prevent network from over fitting. The other way of regularization is L1 or L2 regularization in which we add a term as norm-2 of weights with a hyperparameter *lambda* to prevent weights from being too large. Another way is early stopping, which is going to stop training as soon as the validation error reaches a minimum. Also, we have drop out technique which is going to ignore some units and all of their connections during training and intuitively it reduces the complexity of network, therefore we can deal with overfitting.

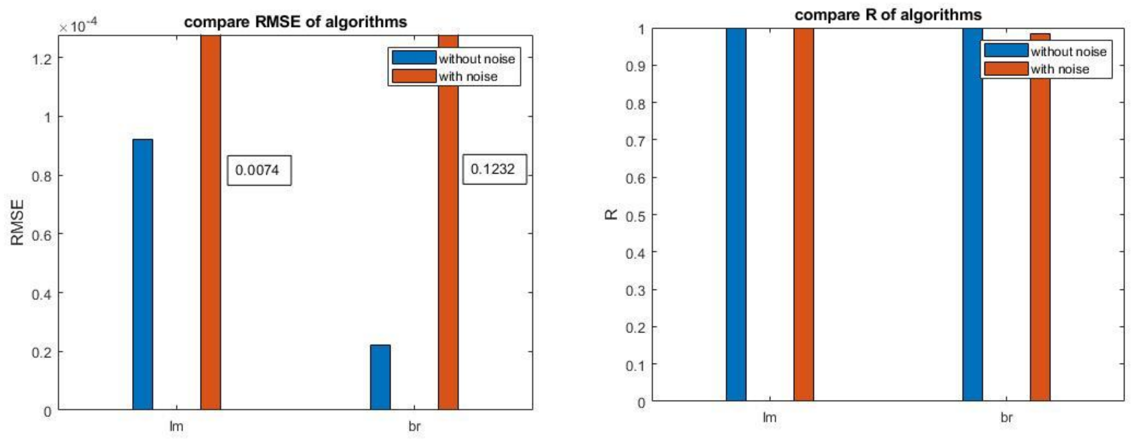


Figure 1.10 RMSE and correlation coefficient of training *lm* and *br* algorithm.

Assignment 2: Applications: Time-series Prediction and Classification

2.1 TIME-SERIES PREDICTION

2.1.1 Santa Fe Dataset

The goal of this exercise is to train a NN on the time series Santa Fe dataset and predict the future observations based on training a model on historical observations. The training and test Santa Fe dataset is shown in Figure 2.1. The prediction for certain time t is equal to a weighted sum of previous values up to a certain lag. The neural network is a multilayer perceptron (MLP) with one hidden layer with feedforward training using Levenberg-Marquardt. The number of epochs for training network is 50 and it is fixed during this experiment and we need to try different combination of lags as p and number of neurons in the hidden layer to select best hyperparameter. The performance is assessed by RMSE (root mean square error).

The first thing is to understand the function of "getTimeSeriesTrainData.m", which nicely divides the training data into batches with size equal to lag. For instance, with Santa Fe training dataset of size 1000 and $lag = p$, the output of this function is a matrix with size $p \times (1000 - p)$, that each column is an input of our NN, and a vector which has the value of the next point after each batch. In Figure 2.2, the first data batch of size $lag = 100$ and the value at $t = 101$ that should be predicted with this batch, are shown.

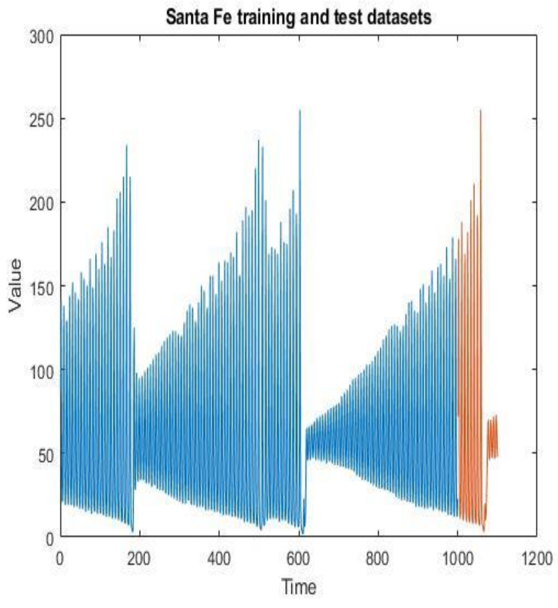


Figure 2.1 Santa Fe training and test datasets.

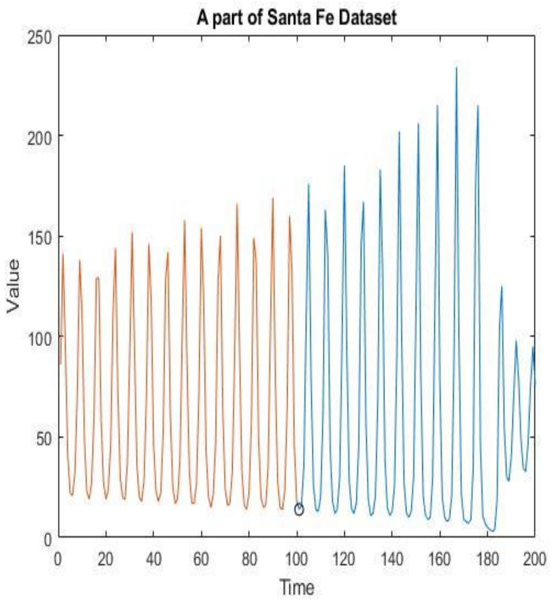


Figure 2.2 First data batch for lag=100.

In the next stage, the network was trained for 50 hidden neurons and different lag values in the interval of 20:10:100. And after that, the network trained for fixed lag value of 100 and number of hidden neurons in the interval of 10:10:50. The training process was done for 5 iterations and at the end the total RMSE was averaged over iterations for each lag values and for each neuron number, in order to consider the fact that in training NN, by choosing different initial values for interconnection weights, we may end up to different local minimum of cost function. As shown in Figure 2.3 and Figure 2.4, it seems that the network performs better by using higher lag values and higher number of hidden neurons, but it is not so monotonic and we can see some irregular point too.

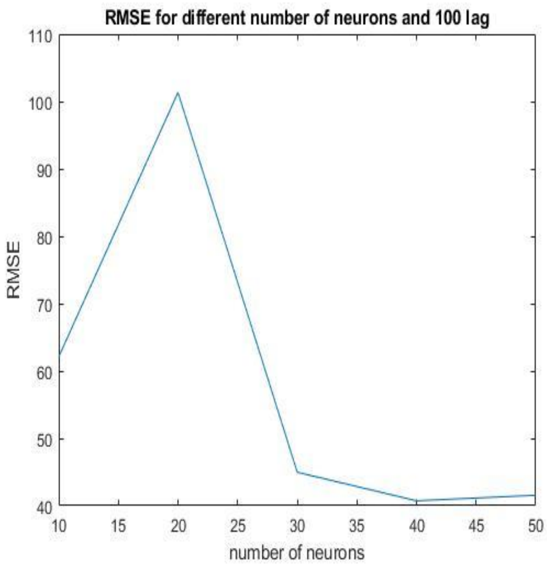


Figure 2.3 RMSE vs number of neurons.

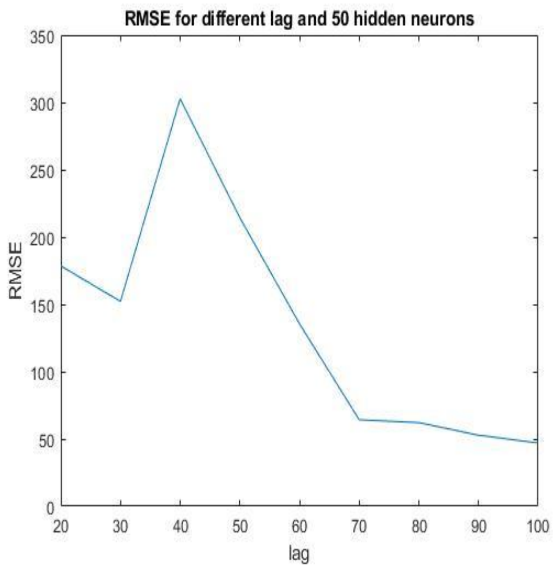


Figure 2.4 RMSE vs lag value.

The best RMSE that we have during all iteration approximately equals to 41 for $lag = 100$ and number of neurons equals to 50. Actually, we also have other predictions with other combination of lag value and number of neurons that perform well in the first 60 point, but after this point it is getting hard for network to follow the correct values. Also, the more we increase the number of lag and hidden neurons, the more the computational power will be needed, and that is why we did the experiment with mentioned intervals for lag and number of neurons.

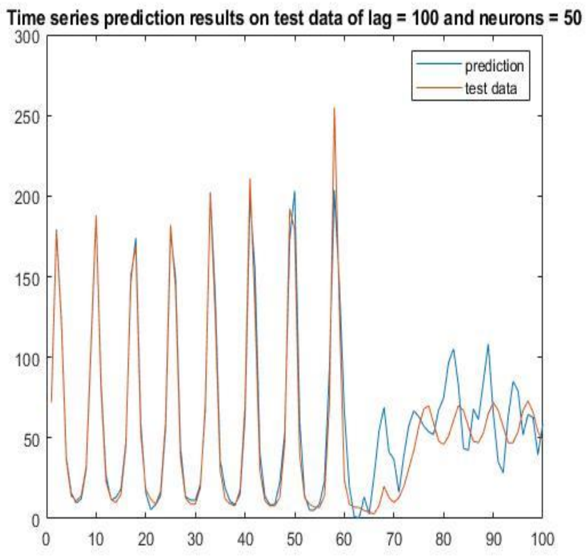


Figure 2.5 Test dataset of time series ans its prediction using trained network.

2.1.2 Climate Change

In this exercise, we are going to train the feedforward neural network on climate time series dataset of one city, using Levenberg-Marquardt method, with one hidden layer, just the same as previous exercise. We consider 80% of data for training and 20% for test. Also, the training set is further split into 80% training and 20% validation. These three datasets are shown in Figure 2.6. The "global_tempreture_train.m" file is run for different combinations of lag values in the interval of 25:5:80 and number of neurons 30 and 50 for 50 epochs. Then, the performance of training is assessed on validation set using MSE (mean square error), which is shown in Figure 2.7. As we can see, there is a trend that by increasing lag, the MSE is decreased but here with 80 lag we still have high error on validation set and still we are overfitting to training set. Perhaps, we need to increase the lag or change the architecture of network, for instance using 2 layers. Based on the figure, we can opt 100 as our lag value and 30 as the number of hidden neurons. The result of prediction on test data set in Figure 2.8 is not satisfactory and after the some starting points that are predicted well, the output of network does not estimate true values correctly and we can see that how small error in the beginning of time series leads to larger error at the end of the series because of error propagation. First assumption is that we need to change the network and use more complex network in order to be able to predict the true time series underlying trend. For instance, we try to have 2 layers with 50 neurons, however the result presented in Figure 2.9 shows that we do not have any improvement and we still have large error.

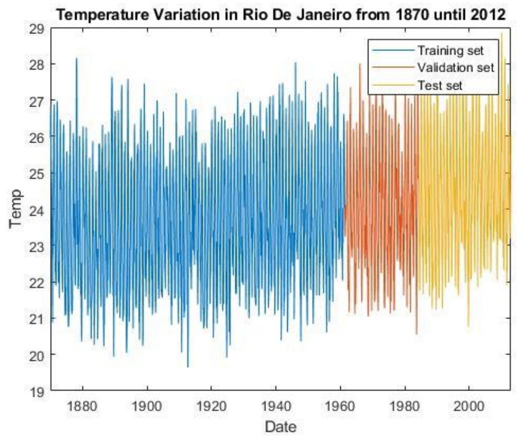


Figure 2.6 Temperature data of Rio De Janeiro

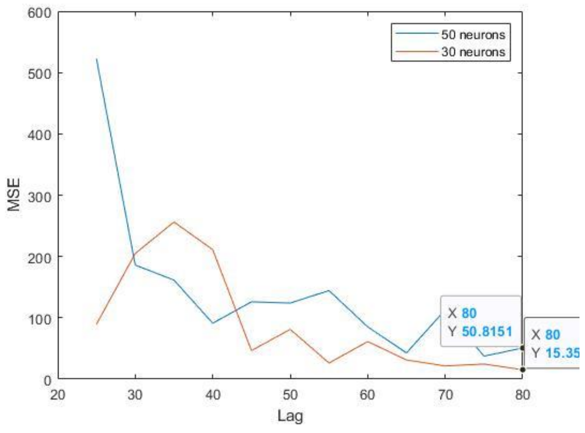


Figure 2.7 Compare the MSE on validation set of models with different neurons and Lag.

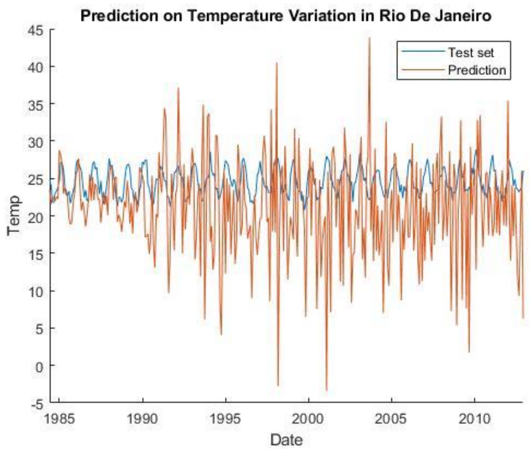


Figure 2.8 Prediction result of test set and true test set, err = 31, 1 layer, 30 neurons.

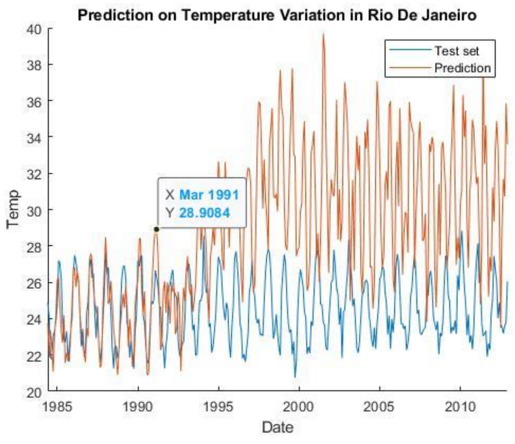


Figure 2.9 Prediction result of test set and true test set, err = 36, 2 layers, 50 neurons

In the next step, we are going to change the algorithm of learning. Instead of Levenberg Marquardt, we are going to use Bayesian Regularization in order to apply regularization and avoid from overfitting, and the network with one layer of 30 neurons and 80 lag. In this method, we do not need validation set and all hyperparameters and regularization coefficient are selected in Bayes framework. From Figure 2.10 we can find that how this regularization leads to outperform the performance of network for predicting new unseen time series and in other word generalize the network.

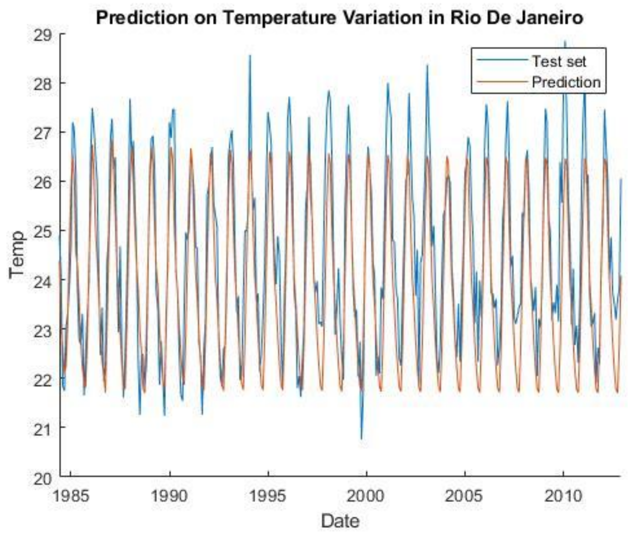


Figure 2.10 Original and predicted data temperature of Rio De Janeiro, using Bayesian regularization, err = 1.48.

2.2 CLASSIFICATION

In this part, we are going to implement a classifier for breast cancer dataset with two classes of malignant and benign cancers. The dataset has 400 datapoint with 30 features or dimensions, so in order to visualize the dataset we need to use *tsne* or *pca* to reduce the dimensionality. We use *tsne* which stands for t-distributed stochastic neighbor embedding and since it minimizes a nonconvex cost function, we need to set the random seed to default. Furthermore, we use 10-fold cross validation. It means that we divide trainset into 10 set and each time take one of sets as our validation set and others for training set and do this procedure until all sets are taken as validation once. Based on the validation set accuracy for different number of neurons and different algorithm, we try to choose the best hyperparameters. We start with a one-layer neural network and epoch number is 100, then try to find the best number of neurons between 20,30,50 and the best algorithm between gradient descent, Levenberg Marquart, and Bayesian regularization, based on the performance on validation set. The performance of classification can be measured using different fractions made by TP (true positive), TN (true negative), FP (false positive), FN (false negative). We use $accuracy = \frac{TP+TN}{TP+TN+FP+FN}$ and $Recal = \frac{TP}{TP+FN}$ and $specificity = \frac{TN}{TN+FP}$. We want to have high recall (sensitivity), since it is so important to have less FN. In our case, it would be dangerous to classify a malignant cancer to a benign cancer (FN). The results are provided in Table 2.1, and we can choose *trainlm* algorithm with 30 neurons. Since we have good result with one layer, we do not add additional layer and increase the complexity of network. In the next step, we train the opted network with training set and validate its performance on test set. in order to prevent network from overfitting, we take 10% of training data as validation set for early stopping. The accuracy and sensitivity in this case is 96% and 98%. The detailed result is provided in Table 2.2.

Table 2.1 compare performance of different algorithms and number of neurons for breast dataset

Algorithm	traingd			traimlm			trainbr		
Neurons	20	30	50	20	30	50	20	30	50
Accuracy	88%	85%	89%	96%	97%	96.5%	95.2%	93%	95.5%
Recall	85%	87%	88%	94%	95%	96%	92%	90%	94%
specificity	14%	10%	9%	3%	1%	3%	3%	4%	3%

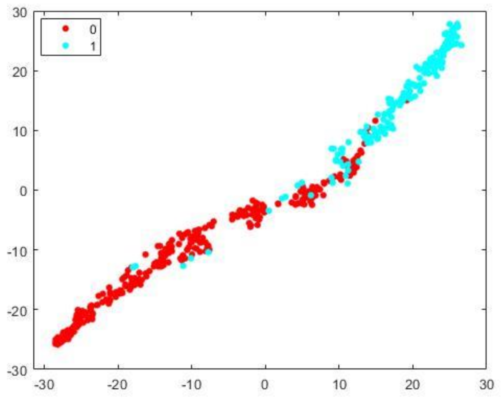


Figure 2.11 Breast cancer dataset visualization.

Table 2.2 Result of classification of test dataset using trained network

	predicted			
Actual	#169	True	False	
	True	TP=61	FN=1	61/62=98%
	False	FP=5	TN=102	102/107=95%
		61/66=92%	102/103=99%	Accuracy=96%

2.3 AUTOMATIC RELEVANCE DETERMINATION

Automatic relevance determination is a technique to find the input features which are more relevant to the output or target. To do this, set of hyperparameters α_i are defined for weights associated with input and the prior over weights has gaussian distribution given by ARD gaussian prior with a variance that is governed by inverse of hyperparameter α_i, therefore, the more relevant features have smaller α_i and large posterior weights. After training network, the hyperparameters are updated using Bayesian re-estimation to get a posterior approximation associated with inputs. Not considering the irrelevant features reduce the complexity of network and therefor time consumption of training.

We apply this method on breast cancer data to find the most relevant features between 30 features to the output. We run the provided MATLAB code "demard.m" for this dataset and put the number of input equal to 30. Then take the first 30 value of *net.alpha* variable and sort them in ascending order to find small hyperparameters and therefore more relevant input features. After that, we train and test our neural network of previous part just for relevant features. We do the experiment for the first 5,8,15,20 most relevant features. The result is provided in Table 2.3. Based on results, the first 15 features are good choice since the performance is near to the performance of no feature reduction.

Table 2.3 Choosing the number of relevant features in breast cancer dataset based on their performance.

#Features	5		8		15		20	
#169	True	False	True	False	True	False	True	False
True	57	5	57	5	59	3	61	1
False	13	94	8	99	7	100	12	95
Accuracy	89%		82%		94%		92%	

Assignment 3: Unsupervised Learning and Data Visualization

3.1 SELF ORGANIZING MAP

3.1.1 SOM on concentric cylinders and banana dataset

In this exercise, we implement SOM with different topology and distance function on concentric cylinders. Before training the network, each point of the lattice has an initial weight which is its position. As we can see in Figure 3.1, after 200 iteration and grid size of $4 \times 4 \times 4$, weight vectors are changed in a way to reach the data topology. The neurons in the layer of an SOM are arranged originally in physical positions according to a topology function. The function gridtop, hextop, or randtop can arrange the neurons in a grid, hexagonal, or random topology. Distances between neurons are calculated from their positions with a distance function. There are four distance functions, dist, boxdist, linkdist, and mandist. Furthermore, in this exercise, we can understand how SOM is similar to vector quantization, since after training, weight vectors are set of prototypes that can represent the whole dataset.

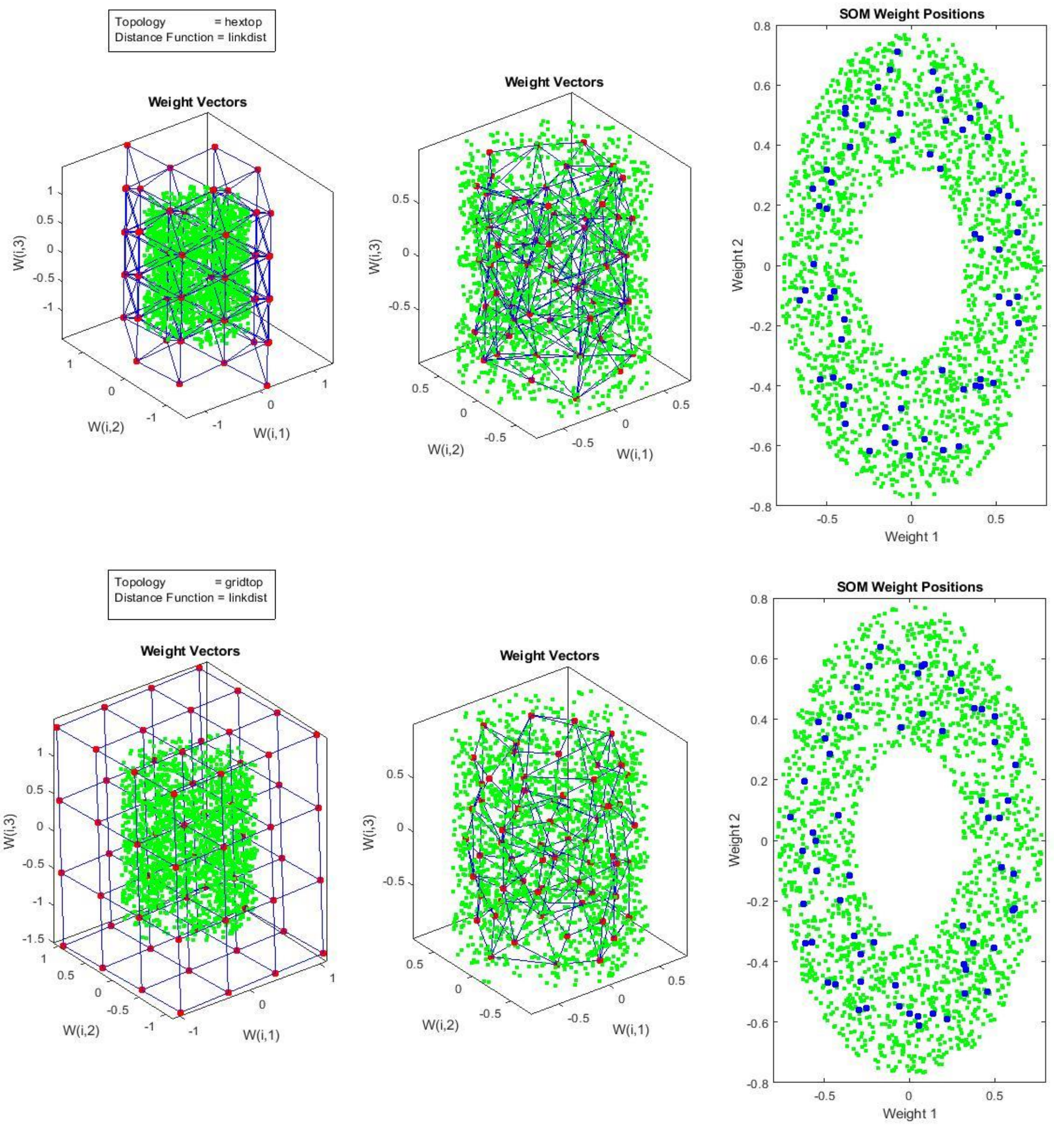


Figure 3.1 SOM for different topology and distance function for concentric cylinders

In the next part, we implement SOM on banana dataset in which we have two separate banana shapes, in two classes. We use grid size of 4×4 for output layer and the number of epochs for training is 400. Before training, weight vectors or prototypes are distributed in feature space based on topology function which is hextop, with initial positions, and after training, as we have in Figure 3.2, prototypes are exactly represent the dataset topology in two dimensions. In this exercise, we can notice how SOM can be used to cluster input vectors according to how they are grouped in the input space. SOM is an unsupervised learning and here we just put labels after training on top of the plot to show we have two classes.

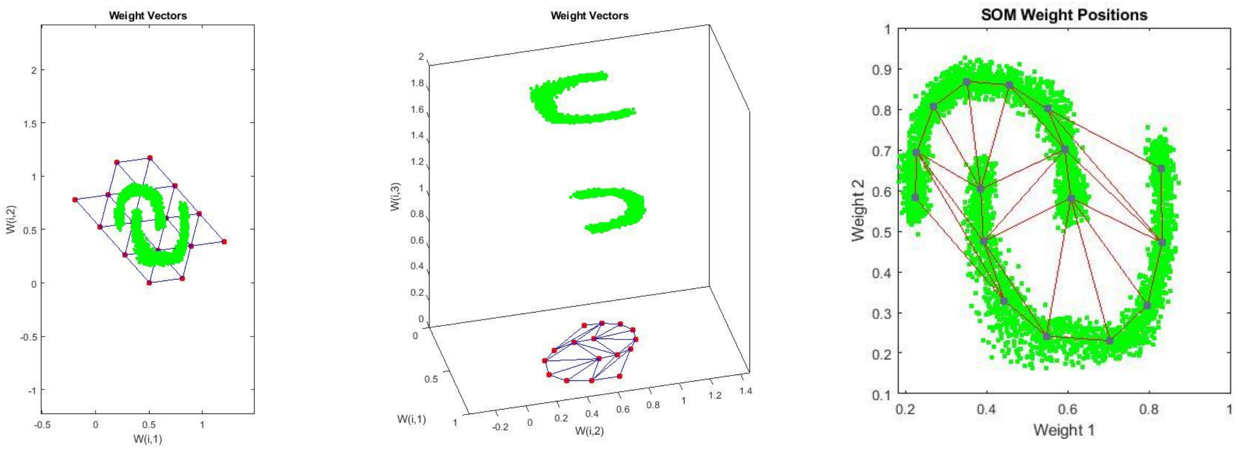


Figure 3.2 SOM on banana dataset.

Besides, we have different plot for weights which have nice interpretations. In figure3.3, there is a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Lighter and darker colors represent larger and smaller weights, respectively.) If the connection patterns of two inputs are very similar, you can assume that the inputs were highly correlated. In this case, input 1 has connections that are very different than those of input 2. Also, in Figure3.4, we have SOM neighbor weight distances in which we can find two groups of connected neurons with lighter color, which are divided from each other with darker color, each one represents one of the two groups of tightly clustered data points. The corresponding weights are closer together in this region.

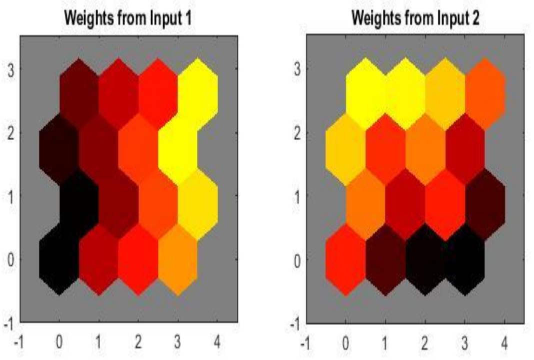


Figure 3.3 SOM input planes

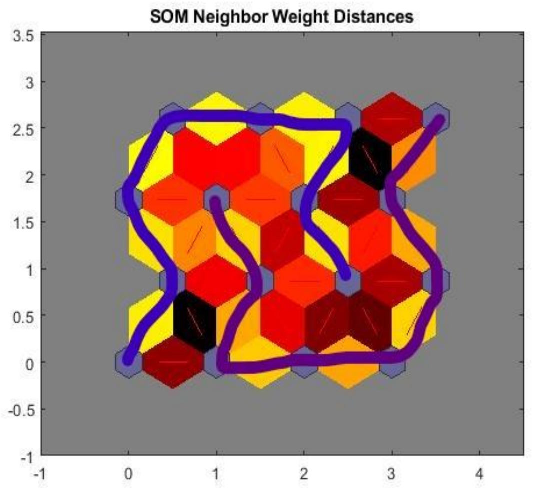


Figure 3.4 SOM weight distances.

3.1.2 SOM on Covertype dataset

In this exercise, we use SOM as clustering method. The dataset has 54 features and 581012 data points. The data was labelled from 1 to 7, so we have 7 different classes. We try a few different numbers for epochs and gridsize and compare the networks and finally we choose network with 100 neurons or 10*10 grid size and 50 as number of epochs.

In order to evaluate the performance, we use *Adjusted Rand Index*. By looking at the true label vector, we understand that we have 7 clusters, but in SOM, we cluster the dataset to $gridsize_x \times gridsize_y$ groups, so we need to compare these two. The definition of ARI is: Given a set *S* of *n* elements(in our example: Covertype dataset with 581012 point), and two clustering of these elements, namely $X = \{X_1, X_2, \dots, X_p\}$ and $Y = \{Y_1, Y_2, \dots, Y_q\}$ (in our example: the X is a true clustering with $p = 7$ and Y is the result of SOM with $q = gridsize_x \times gridsize_y$), the overlap between *X* and *Y* can be summarized in a contingency table $[n_{ij}]$ where each entry n_{ij} denotes the number of objects in common between X_i and Y_j. In "RandIndex.m" function, this index and other three evaluation indexes are implemented. The value of ARI is between -1 and 1. The more it is close to one, the better, meaning that the clustering is near the ground truth that we have. Also, we have Rand index which is a measure of the percentage of correct decisions made by the algorithm, and it is the same as accuracy. Our network with mentioned setup has RI of 0.62.

$X \diagdown Y$	Y_1	Y_2	$\cdots$	Y_s	sums
X_1	n_{11}	n_{12}	$\cdots$	n_{1s}	a_1
X_2	n_{21}	n_{22}	$\cdots$	n_{2s}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_r	n_{r1}	n_{r2}	$\cdots$	n_{rs}	a_r
sums	b_1	b_2	$\cdots$	b_s	

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}\right] \Big/ \binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2}\sum_j \binom{b_j}{2}\right] \Big/ \binom{n}{2}}$$

To visualize the result, we can use neighbor weight distances and SOM sample hits. Groups of light segments appearing in neighbor weight distances, bounded by some darker segments indicate that the network has clustered the data to some groups. Groups can be seen in the Figure 3.4. The corresponding weights (prototypes) are closer together in each group, which is indicated by the lighter colors. Whereas weights in each group are connected to other groups by the darker band in the neighbor distance figure which means that they are far apart. Another useful figure is SOM sample hits that can tell us how many data points are associated with each neurons or in the other word, each prototypes is the nearest one to how many data points.

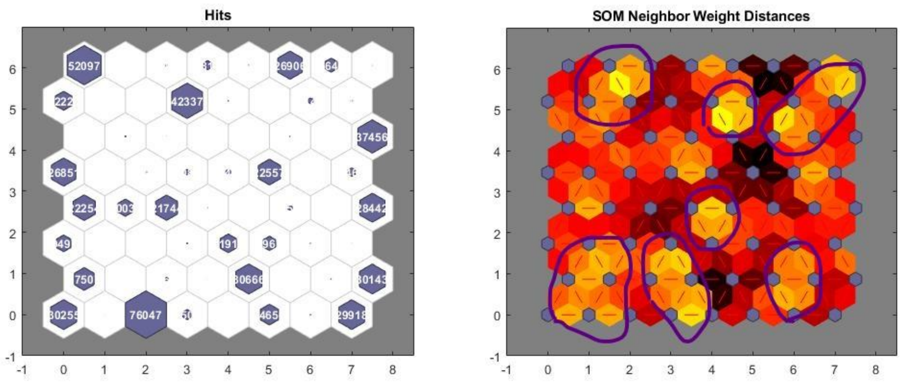


Figure 3.4 Neighbor weight distances and SOM hits for trained network over Cover Type dataset

3.2 PRINCIPAL COMPONENET ANALYSIS

3.2.1,2 PCA for correlated and uncorrelated datasets

We generate a dataset X of 500 data points and 50 dimensions with a random gaussian distribution and by subtracting the mean of the dataset from each data point make it zero-mean. The goal is to apply PCA with reduced dimension of q which is in range of 1 to 49 and compare the reconstructed dataset with the original dataset using root mean square error between the original data set and the reconstructed data set. As we can see in Figure 3.6, the RMSE is decreased linearly by increasing the q.

Then, we do the same experiment on *cloles_all* dataset which is a high correlated dataset with 264 data points and 21 dimensions. This time we get the Figure 3.7 as a relation between RMSE and q. Consequently, we understand that for a correlated dataset since the largest eigenvalue has a large percentage of the sum of all eigenvalues, only a few numbers of eigenvectors or components have a lot of information about the dataset, so we can reduce the dimension significantly without losing important information. However, it is not the case if the dataset is uncorrelated.

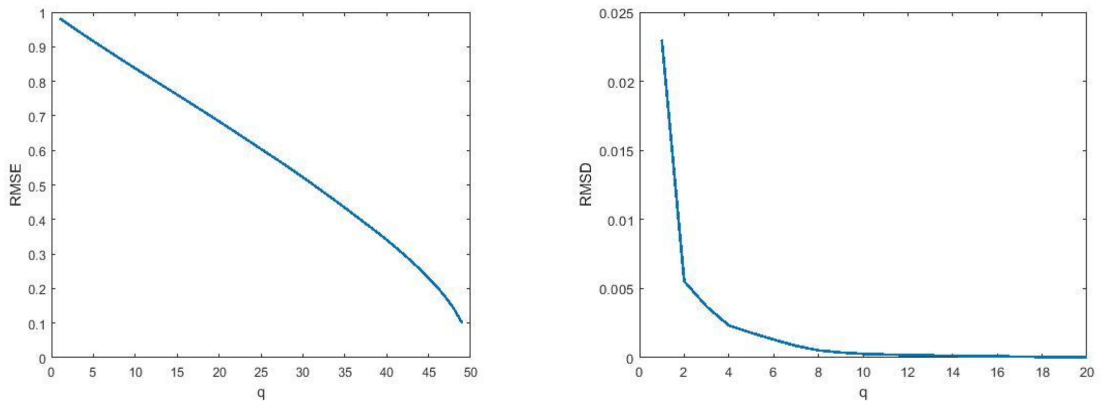


Figure 3.5 RMSE vs number of reduced dimention for gaussian (left) and cloles_all (right) dataset

3.2.3 PCA using mapstd

In this part we implement PCA on *cloles_all* dataset using *mapstd* function to normalize dataset with zero mean and unity variance. After that we use *processpca* command to reduce the dimensionality. This function has an input as *maxfrac* (for example 0.001), means that *processpca* eliminates those principal components that contribute less than 0.001% to the total variation in the dataset. The dimension of dataset is reduced from 21 to 4 and after reconstruction we have RSME equals to 0.0292. we found that we get 1 dimension for $maxfrac = 0.1$ and 21 dimensions for $maxfrac = 10^{-8}$, so by decreasing *maxfrac*, we consider more dimensions so we have less RMSA and we get a figure same as before. But, for gaussian distributed data, if $maxfrac = 0.1$, we get empty output, which means that all of the components contribute less than 0.1% and if $maxfrac = 0.01$, we get the main input, which means that there is no component to contribute less than 0.01% to be eliminated and we cannot find a proper interval for *maxfrac*. It can prove that that the contribution of component in uncorrelated data is similar.

3.3 AUTOENCODER

3.3.1 Image reconstruction using an autoencoder:

A basic autoencoder can be also simply regarded as neural networks, with a basic different which is that autoencoder is consist of encoder and decoder with smaller dimension in comparison to input data dimensions or features, high correlated input feature can be represented in lower dimension. As a result, we're learning compressed representation for approximating input data and not just approximating identity function which remembers original input leading to overfitting. In fact, this is called sparse autoencoder in which we have sparsity penalty in our loss function. In most cases, we construct our loss function by penalizing activations of hidden units so that only a few nodes are encouraged to activate when a single sample is fed into the network. Sparse autoencoder is similar to PCA but for nonlinear dimensionality reduction.

Sparsity regularizer attempts to enforce a constraint on the sparsity of the output from the hidden layer. Sparsity can be encouraged by adding a regularization term that takes a large value when the average activation value, $\hat{\rho}_i$, of a neuron i and its desired value, ρ, are not close in value. One such sparsity regularization term can be the *Kullback-Leibler* divergence which is a function measuring how difference two distribution. In this case, it takes the value zero when ρ and $\hat{\rho}_i$ are equal to each other, and becomes larger as they diverge from each other. Minimizing the cost function forces this term to be small, hence ρ and $\hat{\rho}_i$ to be close to each other. We can define the desired value of the average activation value using the Sparsity proportion which is a proportion of training examples a neuron reacts to. Sparsity proportion is a parameter of the sparsity regularizer. It controls the sparsity of the output from the hidden layer. A low value for Sparsity Proportion usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples. Hence, a low sparsity proportion encourages higher degree of sparsity.

In this exercise, we try different number of hidden units and epochs for sparse autoencoder on digit dataset to find the one with best performance. The reconstruction error is calculated using MSE. After 250 epochs, we do not see any changes in the performance of training set so we fixed the epoch to this number and try different number of neurons. The sparsity regularization is 2 and the sparsity portion is 0.15 for all experiments. Since our goal is image reconstruction, as we expect, if we have more hidden units, we keep more dimensions of input and we have more information from input in bottleneck space and therefore we can reconstruct the image better. The input dimension 28*28 =784 is larger than number of neurons and by sparsity regularization, we avoid autoencoder from overfitting, and with sparsity portion of 0.15 it can mean to some extent that we have 0.15*300=45 active neurons.

Table 3.1 Digit handwritten image reconstruction, autoencoder with different number of epochs and hidden neurons

# Neurons	# Epochs	Training MSE	Test MSE
10	100	0.0384	0.0346
100	500	0.0256	0.0169
200	250	0.0235	0.0145
300	250	0.0227	0.0135

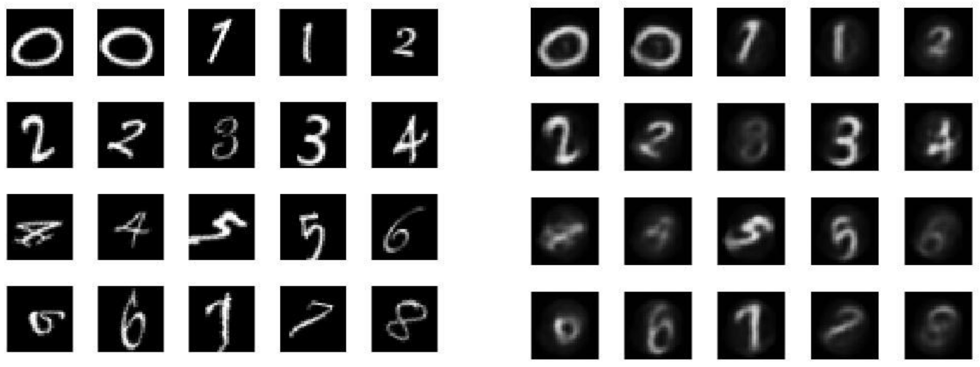


Figure 3.6 Original hand written image(left) and their corresponding reconstruction images with 300 hidden neurons and 250 epochs.

3.4 STACKED AUTOENCODER

In this exercise, we are going to implement stacked autoencoder for classifying digits in images and tune its hyperparameters (number of layers, number of hidden units in each layer, number of epochs). About stacked autoencoder network, we can say it is a MLP (multilayer perceptron) which has an effective way to be trained, training one layer at each time individually instead of training all layers at once, by using sparse autoencoder. It means that we train the first layer and use the result as an input for the second layer and so on. At the end, we stack all autoencoders to form a deep network, and to improve the result, apply finetuning which is performing backpropagation on entire layers. We start with the prepared network in "digitclassification.m" file which has two autoencoder with 100 and 50 neurons, with epoch number of 400 and 100 respectively, and a softmax layer which is trained for a maximum of 400 epochs, in a supervised way to classify the 50-dimensional feature vectors, output of last autoencoder, into 10-digit classes. The number of neurons in each layer is less that the number of inputs in order to learn the compressed representation of the input. In table 3.1, you can find different experiment that we did to tune the hyperparameters. For each of choices, we run the code for 2 iterations *(did not forget to comment rng('default'))* and average the accuracy. It is better to run the network for more iterations but we have limitation on time and computation. The reason for first chois is that, biased on the first layer performance, we can see that it is almost constant after 200 epochs. So, we reduce the epoch for first layer to 250 to make the training faster but the accuracy goes down. Also, we try to add another encoder to the network with 25 neurons, and 50 epochs which cause a decrease in accuracy before fine tuning which is due to the overfitting, and after fine tuning the accuracy is increased but still it is less than before. As we can conclude from the table, decreasing epoch number of first encoder, increasing neurons and epoch of second encoder, and adding one extra layer can not improve the performance of network after fine tuning. Even by increasing the number of neurons in the both first and second layer, although the accuracy without finetuning is getting high, there is no significant improvement in accuracy after finetuning. One important thing is the affect of finetuning which is significantly increase the accuracy.

Table 3.1 hyperparameters tuning of stacked autoencoder network, epochs/ number of neurons

Encoder1	Encoder2	Encoder3	Softmax	No finetuning	finetuning
400 / 100	100 / 50		400 / 10	85.06	99.73
400 / 100	100 / 50	50 E/25 N	400 / 10	41.54	98.54
250 / 100	100 / 50		400 / 10	84.82	99.69
200 / 100	200 / 50		400 / 10	94.94	99.08
250 / 100	100 / 60		400 / 10	89.70	99.62
400 / 150	100 / 60		400 / 10	95.65	99.67

Now, we try to implement a MLP for the same problem (digit images classification) and compare it with stacked autoencoder. In this case, we train the network with one layer, with 5 different number of neurons in the interval of 50:20:130, and we repeat training for each number of neurons for 20 iterations and average the accuracy over all iterations. The accuracy is 96.083, 96.466, 96.477, 97.051, 97.026 for 50, 70, 90, 110, 130 neurons. Also, if we add another layer to the network, the accuracy goes down due to the overfitting. As a result, to compare MLP with stacked autoencoder, we notice that stacked autoencoder with finetuning performs better than MLP for digits classification problem that we have.

Assignment 4: Variational Auto-Encoders and Convolutional Neural Networks

4.1 WEIGHT INITIALIZATION AND BATCH-NORMALIZATION

4.1.1 In this exercise, first of all, we run the *'BatchNormalization.py'* code for training a 6-layer network on a 1000 training examples of CIFAR10 dataset, with and without batch normalization. From the provided image shown in Figure 4.1, we find that, as we expect, **the training converges faster when we use batch normalization**. If the distribution of the inputs to every layer is the same, the network is efficient.

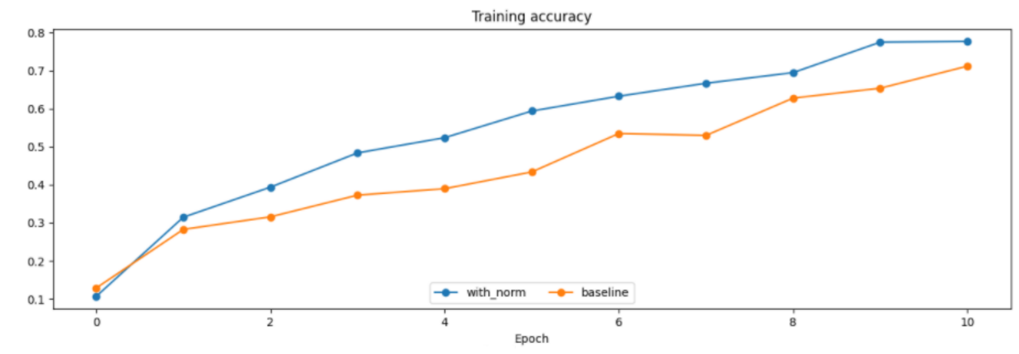


Figure 4.1 Compare accuracy of training with and without batch normalization.

In the next experiment, we train 8-layer network with and without batch normalization and with different scales for weight initialization. Scale is a scalar that gives the standard deviation for random initialization of the weight. Based on Figure 4.2, we can conclude that batch normalization makes weight initialization easier, and the training accuracy is higher when we use batch normalization with the same scale for weight normalization. Weight initialization can be difficult for deep network. Without batch normalization, if weights are initialized with very high or very low values, then $W.x + b$ becomes too high or too small and if an activation function is sigmoid for example, these values are mapped to one or zero where the slope of function changes slowly and learning takes a lot of time. This is referred to vanishing gradient. So, we need to be careful about variance of random initialization of weights which determined the range of values of weights. However, when we use batch normalization, actually we normalized the output of each neuron to the same distribution before applying activation function, therefore the constraints for random initialization of weight vector are relaxed.

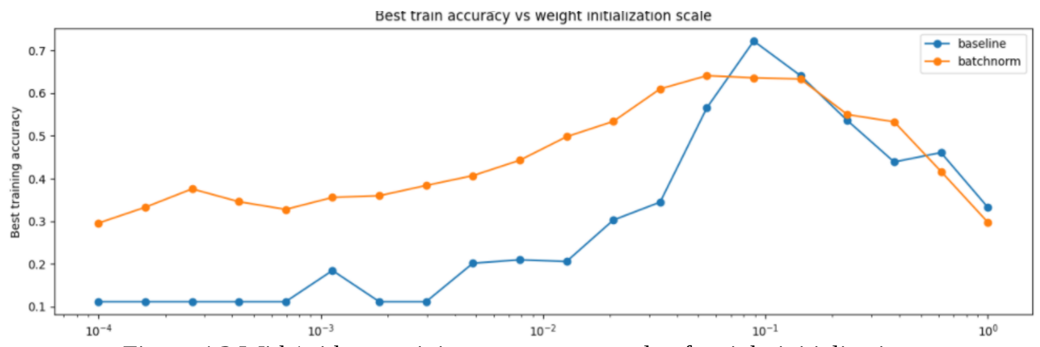


Figure 4.2 With/without training accuracy vs scale of weight initialization.

4.1.2 In this part, we train 6-layer network with and without batch normalization for different batch size. As we can see in Figure 4.3, if we have batch size of 5, the training accuracy with batch normalization is even smaller than the case without batch normalization. Also, by increasing the batch size, we can see that the performance of batch normalization is getting better. The batch normalization has to calculate mean and variance to normalize the output of neurons at each layer across the batch. This statistical estimation will be accurate if the batch size is fairly large and close to the size of real training dataset. while keeps on decreasing as the batch size decreases.

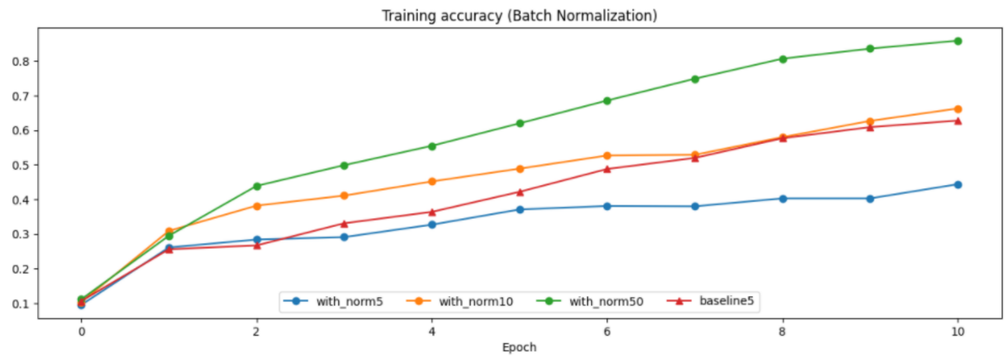


Figure 4.3 Training accuracy for different batch size

4.2 Variational autoencoders

4.2.1 Compare SAE and VAE

The general idea of autoencoders is to learn the best encoding-decoding scheme using an iterative optimization process. So, at each iteration we feed the autoencoder architecture with some data, we compare the encoded-decoded output with the initial data and backpropagate the error through the architecture to update the weights of the networks. Thus, intuitively, the overall architecture creates a bottleneck for data that ensures only the main structured part of the information can go through and be reconstructed. However, dimensionality reduction with no reconstruction loss leads to the lack of interpretable and exploitable structures in the latent space (lack of regularity). As a result, AE cannot be used as a content generator. The relation between AE and content generation is that if the latent space is regular enough, we could take a point randomly from that latent space and decode it to get a new content. The decoder would then act like the generator of a Generative Adversarial Network. The goal with VAE is to regularize the latent space in order to get a continuous and complete distribution of attributes in the latent space and randomly sampling from this distribution will lead to an interpretable content that can be used for content generation. The main similarities between both AE and VAE are that they both consist of encoder and decoder and reduce the number of dimensions of the input data in the bottleneck space, but the structure of bottleneck is different. To put in the nutshell, a variational autoencoder can be defined as being an autoencoder whose training is regularized to avoid overfitting and ensure that the latent space has good properties that enable generative process.

- In VAE, the input in encoded as a distribution over latent space, instead of encoding an input as a single point.
- AE is for purpose of classification, denoising, dimensionality reduction, but VAE is for generating new data.
- The bottleneck region in AE consists of one dimension layer with neurons representing the number of latent, but in VAE we have two-dimensional layer for mean and variance of the sampled point of the latent space.
- Reconstruction error in AE is MSE, which means to minimize the error between input and reconstructed output, but in VSA, we maximize the reconstruction likelihood.
- Sampling from the latent space of AE and decode it does not lead to a meaningful content because we have discontinued distribution of attributes in the latent space, but sampling from the VAE and decode it, by using re-parameterized trick, leads to an interpretable content. Because the latent space is regularized in order to obtain a continuous and complete distribution of attributes in it.
- Regularization in AE is done by adding sparsity constraint in order to prevent network from memorizing the inputs, but in VAE it is done to ensure that sampling from latent space leads to generate content, by means of $p(z|x)$ to have normal distribution.

4.2.2 Optimizer of SAE and VAE

The optimizer for stacked autoencoder in previous exercise is *Scaled Conjugate Gradient* and for variational autoencoder is *Adam Optimizer*.

- In gradient descent, we use line search method and, in each iteration, we find the direction of descent and move along that direction with a proper step size. In the conjugate gradient algorithms, a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. Also, especially scaled conjugate gradient is designed to avoid the time-consuming line search method.

- Adam is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning.

4.3 Convolutional Neural Networks

4.3.1

Each row in A is a shifted version of previous row, so we can implement this convolution using matrix multiplication as follows:

$$y_{4\times1} = A_{4\times6}x_{6\times1}\,, y = \begin{bmatrix}6\\6\\7\\8\end{bmatrix}, x = \begin{bmatrix}2\\5\\4\\1\\3\\7\end{bmatrix}, k = \begin{bmatrix}1\\0\\1\end{bmatrix}, stride = 1\,, A = \begin{bmatrix}1&0&1&0&0&0\\0&1&0&1&0&0\\0&0&1&0&1&0\\0&0&0&1&0&1\end{bmatrix}$$

We cannot reconstruct x by just multiplication of each side with transpose of A, because A is not a unitary matrix that its inverse is equal to its transpose. Also, A is not square, so its inverse is computed using pseudo-inverse method. Put all these aside, here we have underdetermined set of equations (larger number of unknown than number of equations) which has infinitely many solutions and we cannot reconstruct x exactly. What we can is to add our observation, which means that suppose that x is zero padded and we have 2 more element for y at first and end of the vector. In such a case, we have overdetermined set of equations and we can solve it using least square minimization which is $\min_{x} \|y - Ax\|_2^2$.

$$A^T = \begin{bmatrix}1&0&0&0\\0&1&0&0\\1&0&1&0\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{bmatrix}, \quad A^T y = \begin{bmatrix}6\\6\\13\\14\\7\\8\end{bmatrix} \neq x$$

$$y_{8\times1} = A_{8\times6}x_{6\times1}\,, y = \begin{bmatrix}2\\5\\6\\6\\7\\8\\3\\7\end{bmatrix}, x = \begin{bmatrix}2\\5\\4\\1\\3\\7\end{bmatrix}, k = \begin{bmatrix}1\\0\\1\end{bmatrix}, A = \begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\1&0&1&0&0&0\\0&1&0&1&0&0\\0&0&1&0&1&0\\0&0&0&1&0&1\\0&0&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}, \qquad x = A\backslash y$$

4.3.2 Image classification using pre-trained CNN *AlexNet*

i. The pre-trained CNN, *AlexNet*, consists of 23 layers which five of these are convolutional layers followed by a *ReLu* layer. It takes a color image of size 227*227*3 as input and classifies that image as one of the 1000 classes (we only consider 3 classes in this excersise). The feature extraction is done in convolutional layers. The first convolutional layer weight dimension is 11*11*3*96. This means that we have 96 convolutional kernels of size 11*11*3, these convolutional kernels(filters) learned by the network is visualized in Figure 4.4 where we can see the 96 features that are extracted in the first convolutional layer.

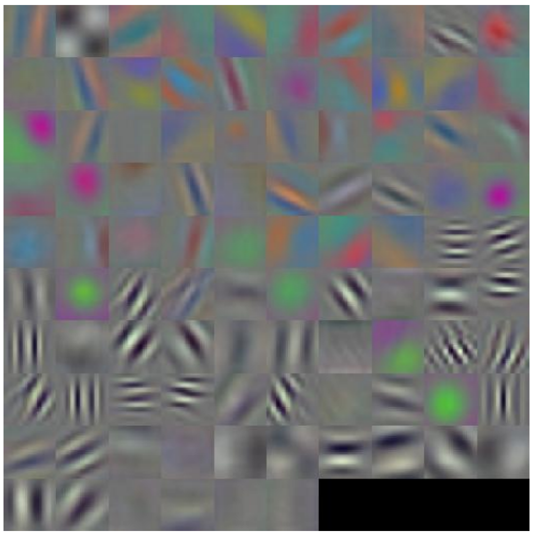
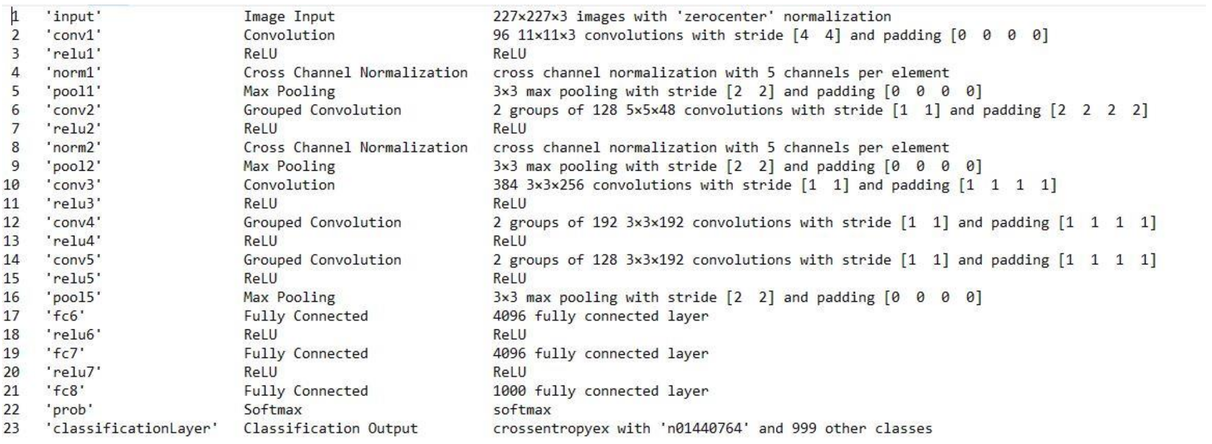


Figure 4.4 First convolutional layer weights.

ii. Inorder to find the dimension of the input of the layer 6, we need to find the output dimension after applying 96 11*11*3 filters with stride [4 4] and zero padding [0 0 0 0]. When four values are specified for padding, the paddings apply to the top, right, bottom, and left in that order (clockwise), and when two values are specified for stride, shows the stride value at x and y direction of image. By using formula $\left\lfloor\frac{n+2p-f}{s}+1\right\rfloor$ $, s = stride = 4, p = padding = 0, f = kernel(filter) = 11$, the output of convolution with input size of 227*227*3 has the size of $\left\lfloor\frac{227+2(0)-11}{4}+1\right\rfloor = 55$. So, we have 96 images of size 55*55. After that we have 3*3 max-pooling with stride [2 2] and padding [0 0 0 0]. We apply the same formula, and we have $\left\lfloor\frac{55+2(0)-3}{2}+1\right\rfloor = 27$. As a result, the input size of layer 6 is 27*27*96.



```
1    'input'                Image Input                    227×227×3 images with 'zerocenter' normalization
2    'conv1'                Convolution                    96 11×11×3 convolutions with stride [4  4] and padding [0  0  0  0]
3    'relu1'                ReLU                           ReLU
4    'norm1'                Cross Channel Normalization    cross channel normalization with 5 channels per element
5    'pool1'                Max Pooling                    3×3 max pooling with stride [2  2] and padding [0  0  0  0]
6    'conv2'                Grouped Convolution            2 groups of 128 5×5×48 convolutions with stride [1  1] and padding [2  2  2  2]
7    'relu2'                ReLU                           ReLU
8    'norm2'                Cross Channel Normalization    cross channel normalization with 5 channels per element
9    'pool2'                Max Pooling                    3×3 max pooling with stride [2  2] and padding [0  0  0  0]
10   'conv3'                Convolution                    384 3×3×256 convolutions with stride [1  1] and padding [1  1  1  1]
11   'relu3'                ReLU                           ReLU
12   'conv4'                Grouped Convolution            2 groups of 192 3×3×192 convolutions with stride [1  1] and padding [1  1  1  1]
13   'relu4'                ReLU                           ReLU
14   'conv5'                Grouped Convolution            2 groups of 128 3×3×192 convolutions with stride [1  1] and padding [1  1  1  1]
15   'relu5'                ReLU                           ReLU
16   'pool5'                Max Pooling                    3×3 max pooling with stride [2  2] and padding [0  0  0  0]
17   'fc6'                  Fully Connected                4096 fully connected layer
18   'relu6'                ReLU                           ReLU
19   'fc7'                  Fully Connected                4096 fully connected layer
20   'relu7'                ReLU                           ReLU
21   'fc8'                  Fully Connected                1000 fully connected layer
22   'prob'                 Softmax                        softmax
23   'classificationLayer'  Classification Output          crossentropyex with 'n01440764' and 999 other classes
```

Figure 4.5 CNN architecture of AlexNet.

iii. We use same formula for all the convolutions and max-pooling layers until the first fully connected layer:

1-5: $\left\lfloor \frac{227+2(0)-11}{4} + 1 \right\rfloor = 55\,, \quad \left\lfloor \frac{55+2(0)-3}{2} + 1 \right\rfloor = 27 \;\; \rightarrow \;\; 27 \times 27 \times 96$

6-9: $\left\lfloor \frac{27+2(2)-5}{1} + 1 \right\rfloor \quad = 27\,, \quad \left\lfloor \frac{27+2(0)-3}{2} + 1 \right\rfloor = 13 \;\; \rightarrow \;\; 13 \times 13 \times 128 \times 2$

10: $\left\lfloor \frac{13+2(1)-3}{2} + 1 \right\rfloor \quad = 13\,, \;\; \rightarrow \;\; 13 \times 13 \times 384$

12: $\left\lfloor \frac{13+2(1)-3}{2} + 1 \right\rfloor \quad = 13\,, \;\; \rightarrow \;\; 13 \times 13 \times 192 \times 2$

14-16: $\left\lfloor \frac{13+2(1)-3}{2} + 1 \right\rfloor \quad = 13\,, \quad \left\lfloor \frac{13+2(0)-3}{2} + 1 \right\rfloor \;\; = 6 \;\; \rightarrow \;\; 6 \times 6 \times 256$

So, the input of first fully connected layer has $6 \times 6 \times 256 = 9{,}216$ neurons and if we compare this with original input size $227 \times 227 \times 3 = 154{,}587$, we find a significant reduction which help us to reduce the number of parameters in classification. Since we have 1000 classes, the softmax layer has 4096 input and 1000 outputs neurons and classification layer uses the probabilities returned by the softmax activation function for each input to assign it to one of the mutually exclusive classes.

4.3.3 Small CNN on handwritten digits dataset

In this exercise, we run the provided code, CNNDigits, and try to improve the performance. Intuitively, if we increase the number of filters, we can extract more features from the image and therefore we get a better performance whereas consume more time. Also, as we go deeper in our network and add layers, the complexity of features is increased and we have more high-level features such as contours. However, here if we increase number of layers from 2 to 3, it does not enhance accuracy (even decrease it), and the intuition behind that can be refer to this fact that our images are not too complicated and do not consist of high-level features, so adding another layer is not useful. Also, the size of filter can be seen as receptive field in CNN and it means how many neighbors' information from image we can see and its choice is mostly related to the dataset. By, increasing the filter size, the number of parameters increase quadratically and therefor we prefer smaller numbers.

Conv1	12 5*5	20 5*5	32 5*5	32 5*5	32 3*3	32 8*8	32 5*5
Conv2	24 5*5	40 5*5	-	64 5*5	64 3*3	64 8*8	64 5*5
Conve3	-	-	-	-	-	-	128 5*5
Time	31	37	35	50	54	71	68
Accuracy	81%	95%	97%	98.4%	97%	96.7%	98%

REFERENCES:

1. Slides, lecture notes.
2. https://nl.mathworks.com/help/deeplearning/gs/cluster-data-with-a-self-organizing-map
3. https://nl.mathworks.com/help/deeplearning/ref/trainautoencoder
4. https://en.wikipedia.org/wiki/Rand_index
5. https://en.wikipedia.org/wiki/Cross-validation_(statistics)
6. https://towardsdatascience.com/build-the-right-autoencoder-tune-and-optimize-using-pca-principles-part-i
7. https://towardsdatascience.com/how-to-make-an-autoencoder
8. https://towardsdatascience.com/understanding-variational-autoencoders-vaes
9. https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks
10. https://towardsdatascience.com/what-is-batch-normalization
11. https://medium.com/temp08050309-devpblog/dl-5-weight-initialization-and-batch-normalization
12. https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-
13. https://towardsdatascience.com/understanding-variational-autoencoders-vaes
14. http://primo.ai
15. https://arxiv.org/abs/1412.6980