**NEURAL NETWORKS**

**Assignment 1 (Group 4)**

**Group Members:**

Akshay Miterani (201401443)

Deep Raiya (201401221)

Anusha Phadnis (201401098)

Mit Naria (201401448)

Mihir Limbachia (201401456)

Tanmay Patel (201401409)

Urmil Kadakia (201401013)

**Objective:**

→ The objective of this assignment is to assess different approaches for doing function approximation and classification using Neural Networks and to analyze different factors that affect the performance of the network.

**Approaches used:**

→ Here, we have used two approaches for function approximation (1)Multi Layer Perceptron With with Gradient Descent & LS and (2)Radial Basis Functions with Gradient Descent & LS).

**Function Approximation:**

→ Neural networks can very effectively be used for function approximation, using datasets that have a training set containing the known mappings between the dependent and independent variables for the function. The network will then learn to create mappings between the variables, if size of the training dataset is big enough and enough number of epochs are used to reduce the error in approximation.

**Approaches and results:**

→ Multi Layer Perceptron :

→ Radial Basis Function:

**MLP\_LS Approximation :**

→ In this approach , two layer MLP is used.The activation function used for hidden layer is log sigmoid and the output layer is purelin. The outputs are predicted function values.The activations functions are :

Log sigmoid:

sigmoid.PNG

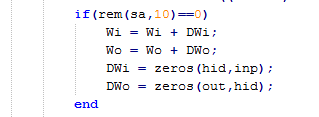
→ The weight matrix for the hidden layer and the output layer of the two layer perceptron used are assigned randomly and for different problems, we use different approaches for their updates, like

(1) For Problem “BJ” batch training is done on all the input data in each individual epoch and total change in the weight matrix for each layer is calculated on the basis of error calculated by loss function. Finally the weight matrix is updated and used for next epoch.

(2) For Problem “SI” : It has very large training data set, so if we sum all the error and then by fixing learning rate very small value(around 1e-4) we update the weights, but its take very much time to converse, Instead we use sequential weights update, where we are doing error calculation for each and every sample and instantly updating weights for the same and and use updated weights for next coming samle.

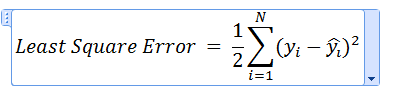
(3) For Problem “MG85”: It has not very large not very small , around 3000 sample training data. So first we have tried Batch update approach, which give us RMSE of 2\*1e-1 ‘s order, So we use partial batch update. What we meant by partial batch update is that fr certain no of training samples (here for particular for this problem 10 samples) we calculate sum of error and update weights for this error and use updated weights for next 10 Sample training data and again calculate the error and doing so we get 1\*1e-1(which is half from fully batch update) and also it converges fast.

Code snippet for the same is as below :



→ The loss function used is:

Least Squared :



→ The loss function is minimised through gradient descent learning using following equations:

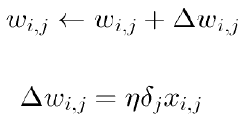
For hidden layer:

hiddenLayer.PNG

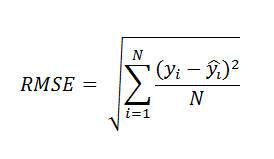
For output layer:

Outputlayer.PNG

And the weights are updated as follows using following equations:



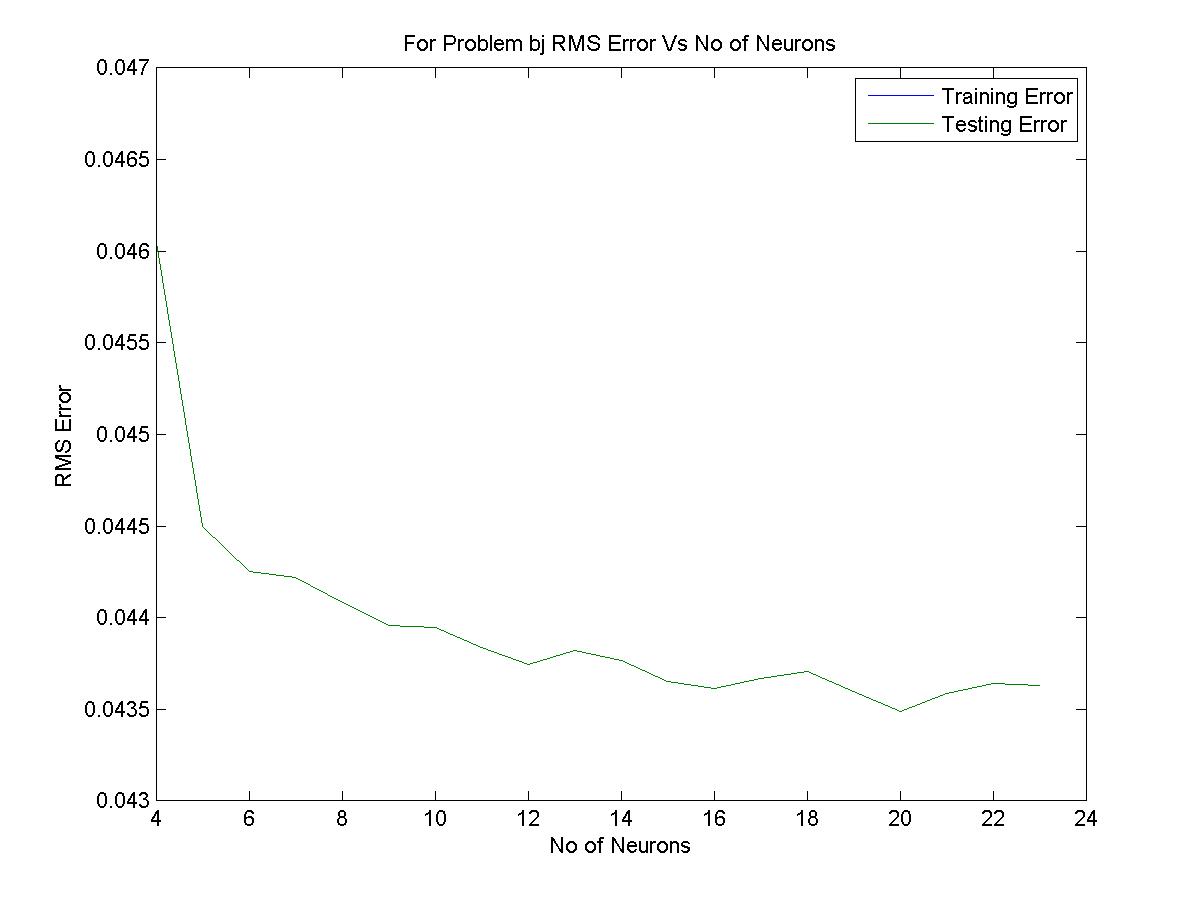
→ After network is trained for training data , the RMSE(Root Mean Square Error)are calculated while validating the training data and during class prediction for the test data.



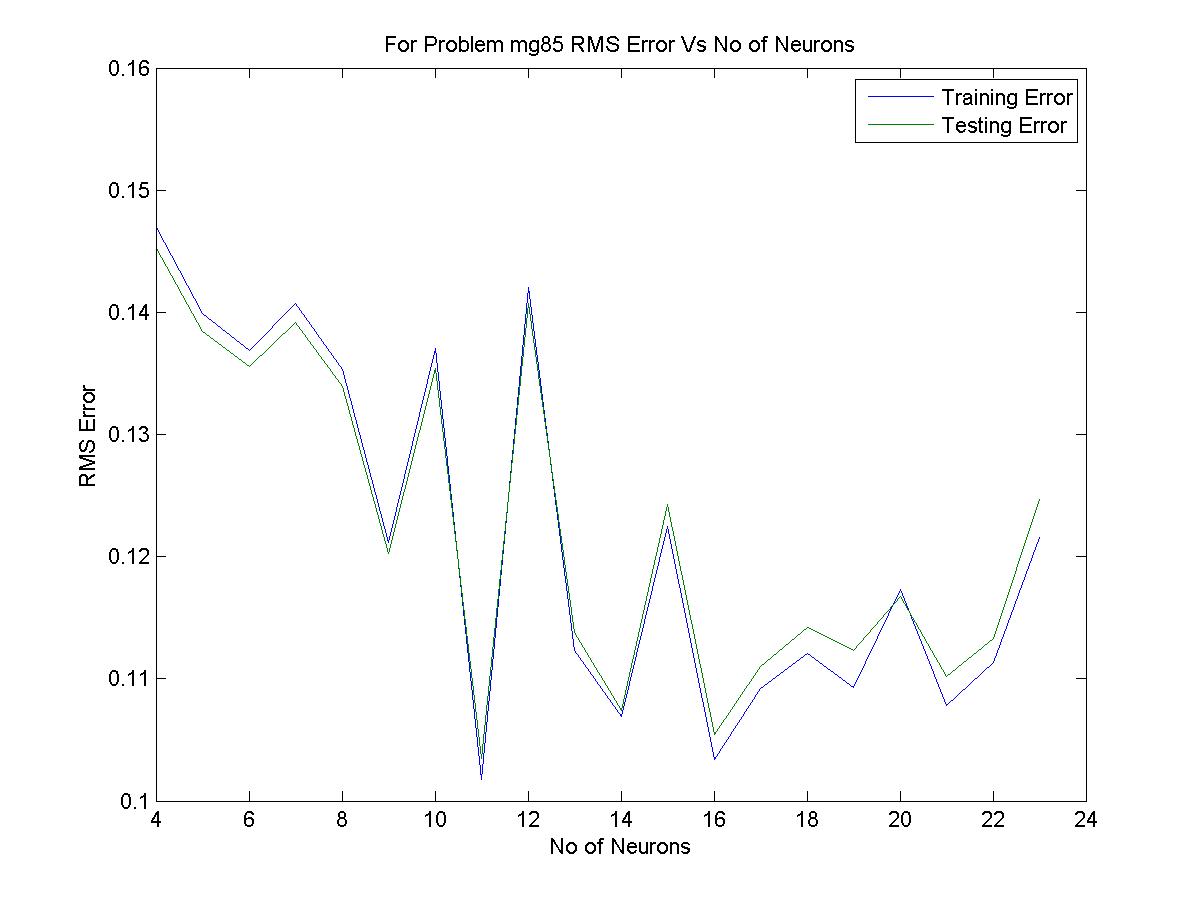
→ The no of hidden layer neurons and no of epochs are selected taking maximising the average accuracy into consideration.

→ Following graph shows how the test and training data average accuracy varies with respect to no of hidden neurons for epoch set to 2000 for Problem BJ with LS loss function. The RMSE not much decreased after hidden neurons = 5. It decreases of order of 0.005’s order, which we don’t think appropriate to double no. of neurons for this much improvisation. Same procedure follows for all problems and graph for same are below.

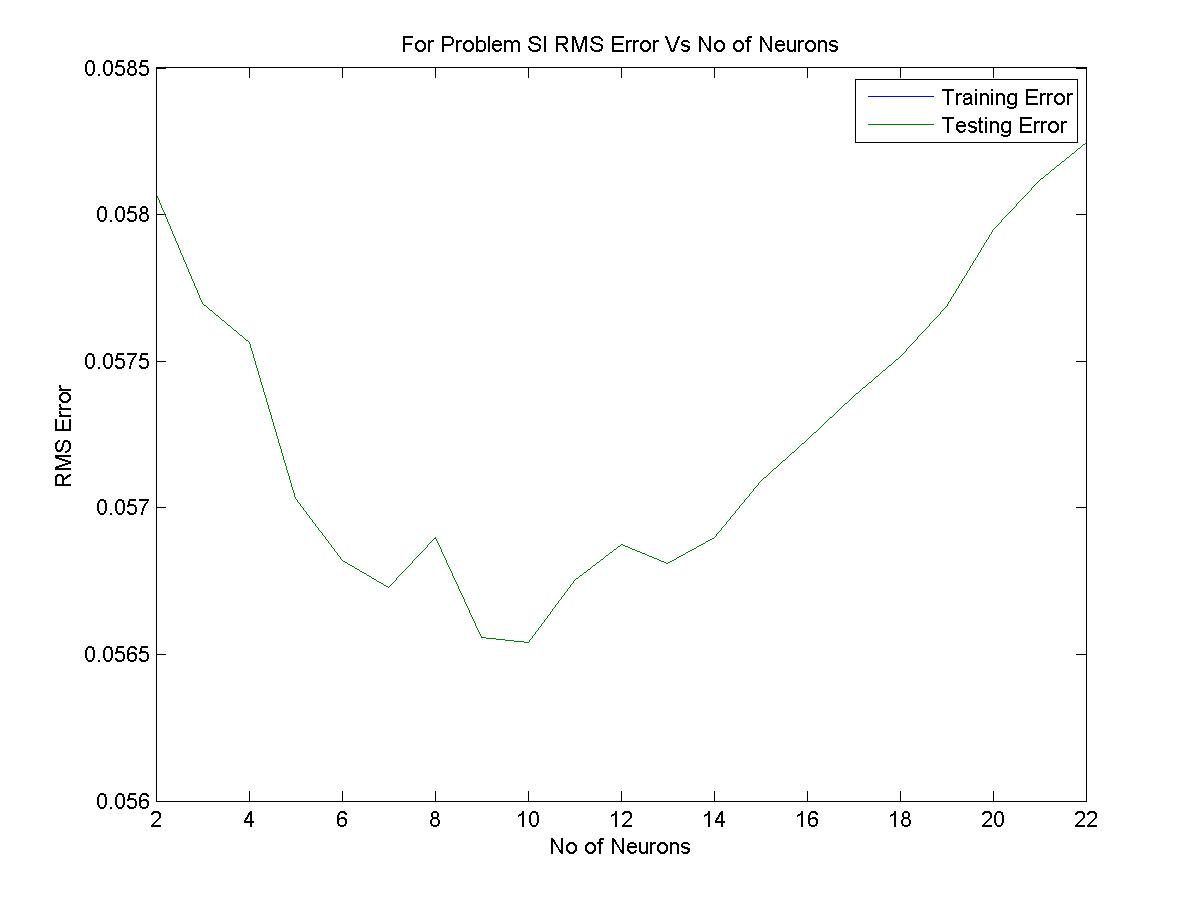
1. For Problem “BJ”

.

(2) For Prob “MG85” :



(3) For Problem SI :

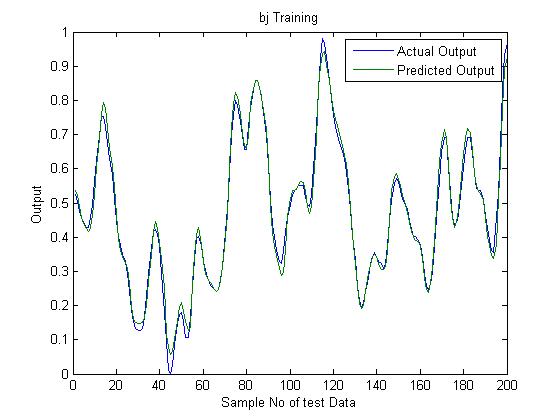


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Problem Name | No of Neurons | Epoch | Learning Rate | Training RMSE Error | Testing RMSE Error |
| BJ | 4 | 2000 | 0.01 | 0.023212 | 0.046148 |
| MG85 | 11 | 1200 | 0.01 | 0.1017 | 0.1018 |
| SI | 10 | 20 | 0.01 | 0.0563 | 0.0563 |

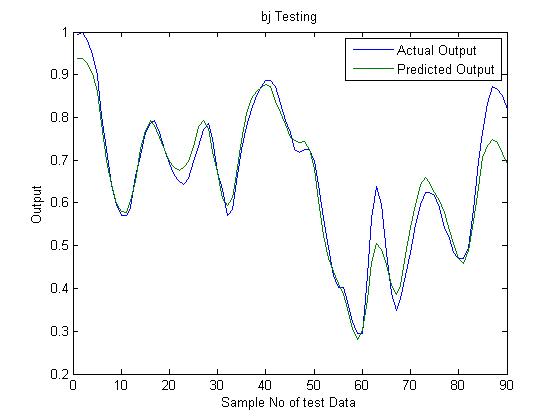
**Actual Output Vs Predicted Output Comparison :**

**→** Below some more graphs for predicted output and actual outputs comparison for every problem for training and testing.

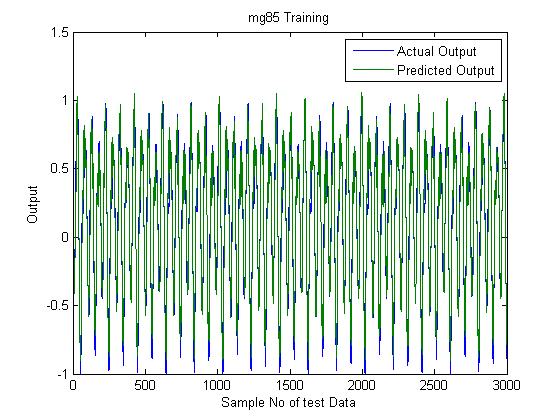
1. For Problem BJ :
   1. Training :



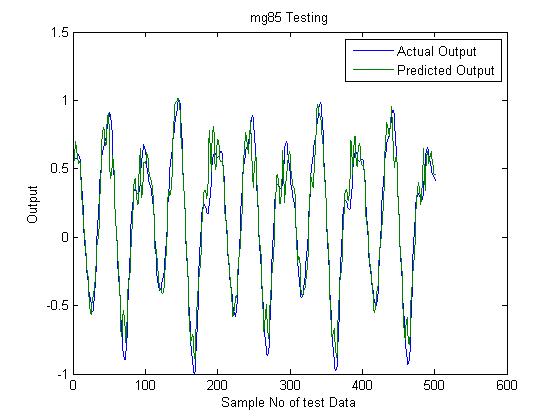
* 1. Testing :



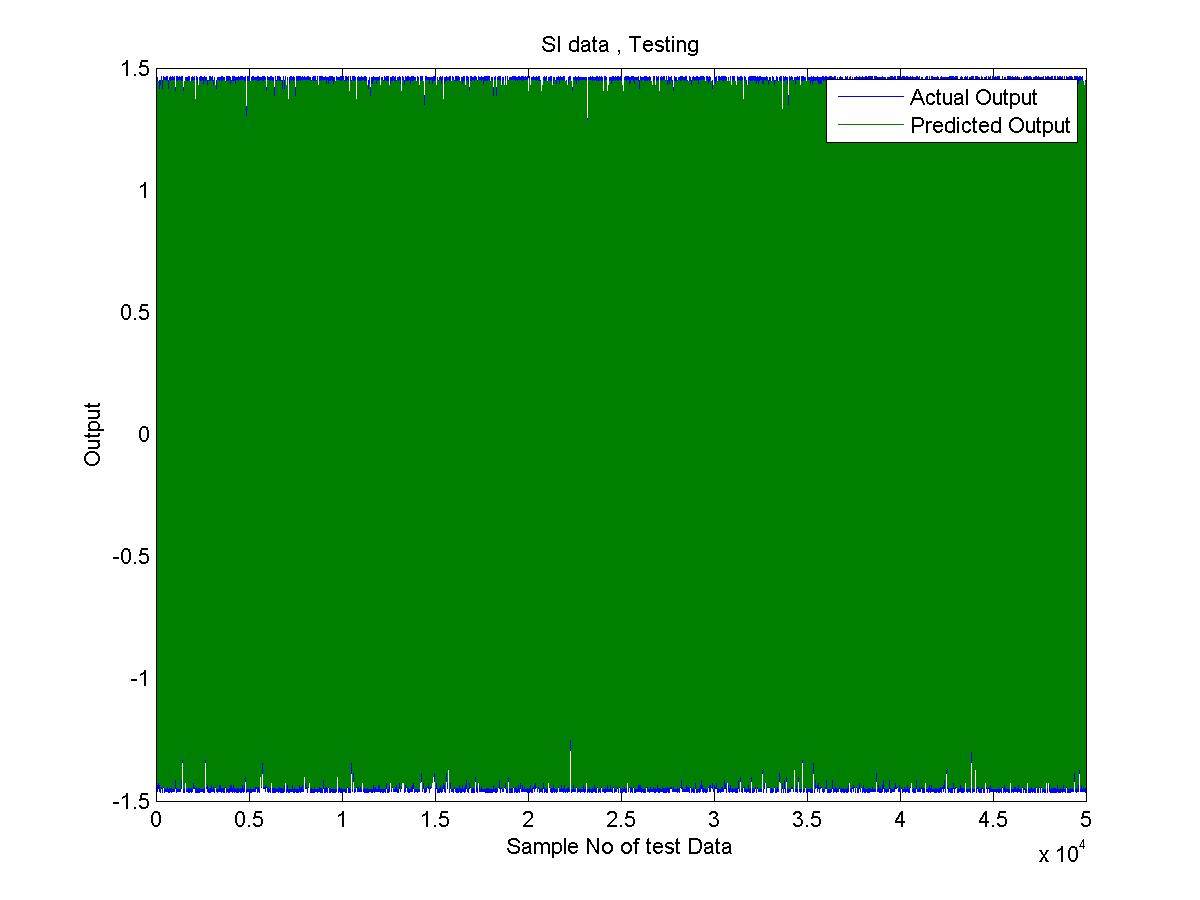
1. For Problem MG85 :
   1. Training :



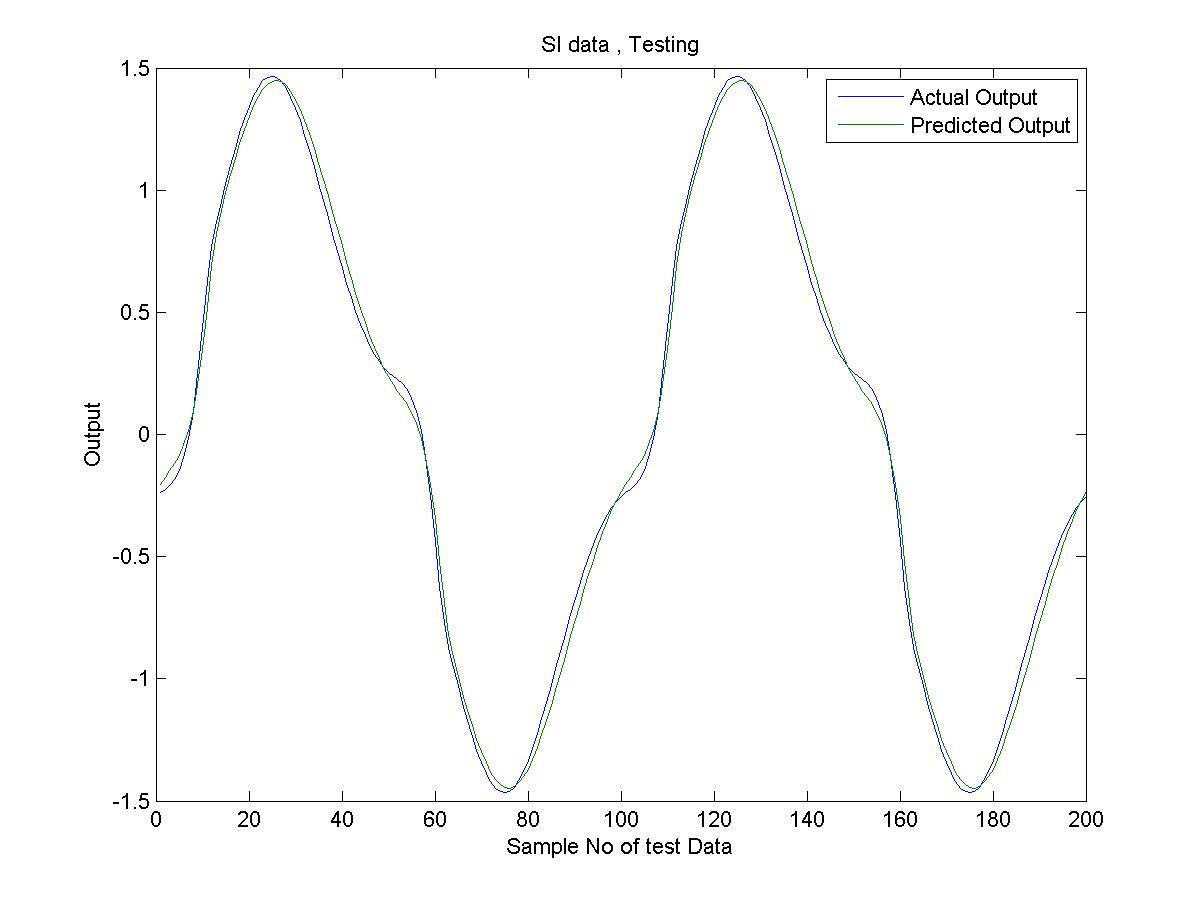
* 1. Testing :



1. For Problem SI :
   1. Training :



* 1. Testing :



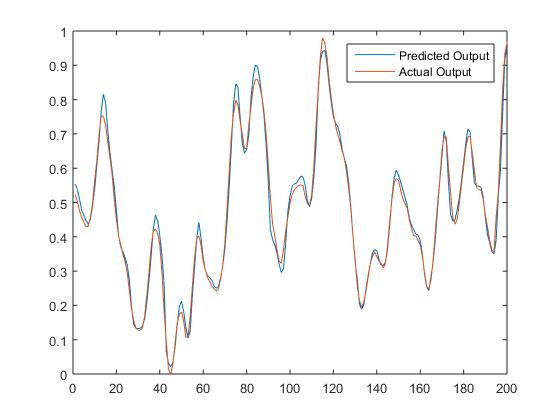
**(2) RBF :**

In RBF, radial basis functions are used as activation functions in neurons. First we initialize neurons with random data points selected from input and we also initialize spreads with near zero random values. Then , we train network by feeding it data and then optimizing spreads, weights and sigmas through gradient descent method.

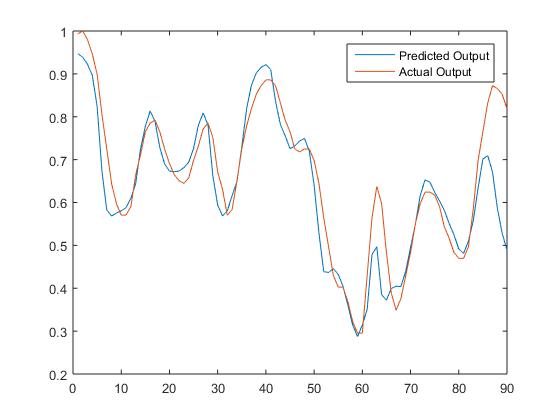
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Problem Name | No of Neurons | Epoch | Learning Rate for weights | Learning Rate  For centres | Learning rate for sigmas | Training RMSE Error | Testing RMSE Error |
| BJ | 4 | 800 | 0.01 | 0.001 | 0.0001 | 0.0602 | 0.0602 |
| MG85 | 5 | 500 | 0.01 | 0.001 | 0.0001 | 0.1078 | 0.302 |
| SI | 4 | 100 | 0.01 | 0.001 | 0.0001 | 0.0173 | 0.0173 |

**Actual Output Vs Predicted Output Comparison :**

**For Problem BJ:**

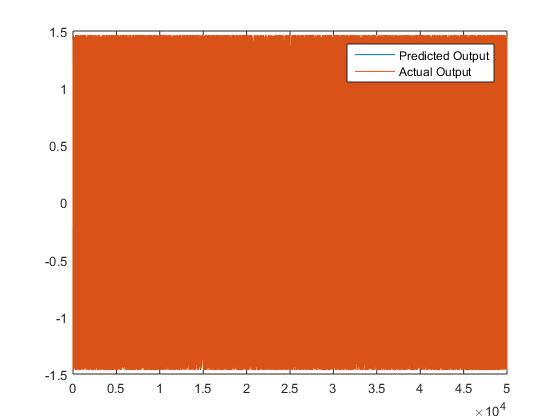
(a) training : Function value vs sample no: 

(b) testing : function value vs sample no:

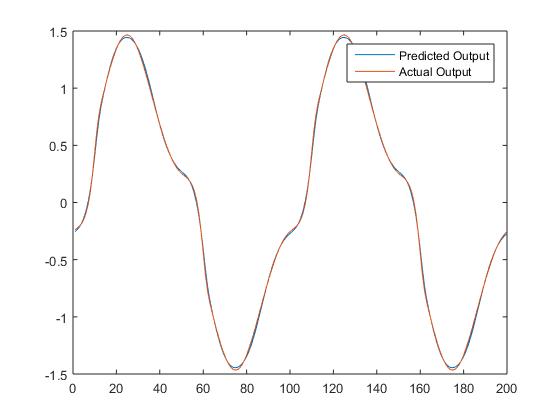


**For Problem SI:**

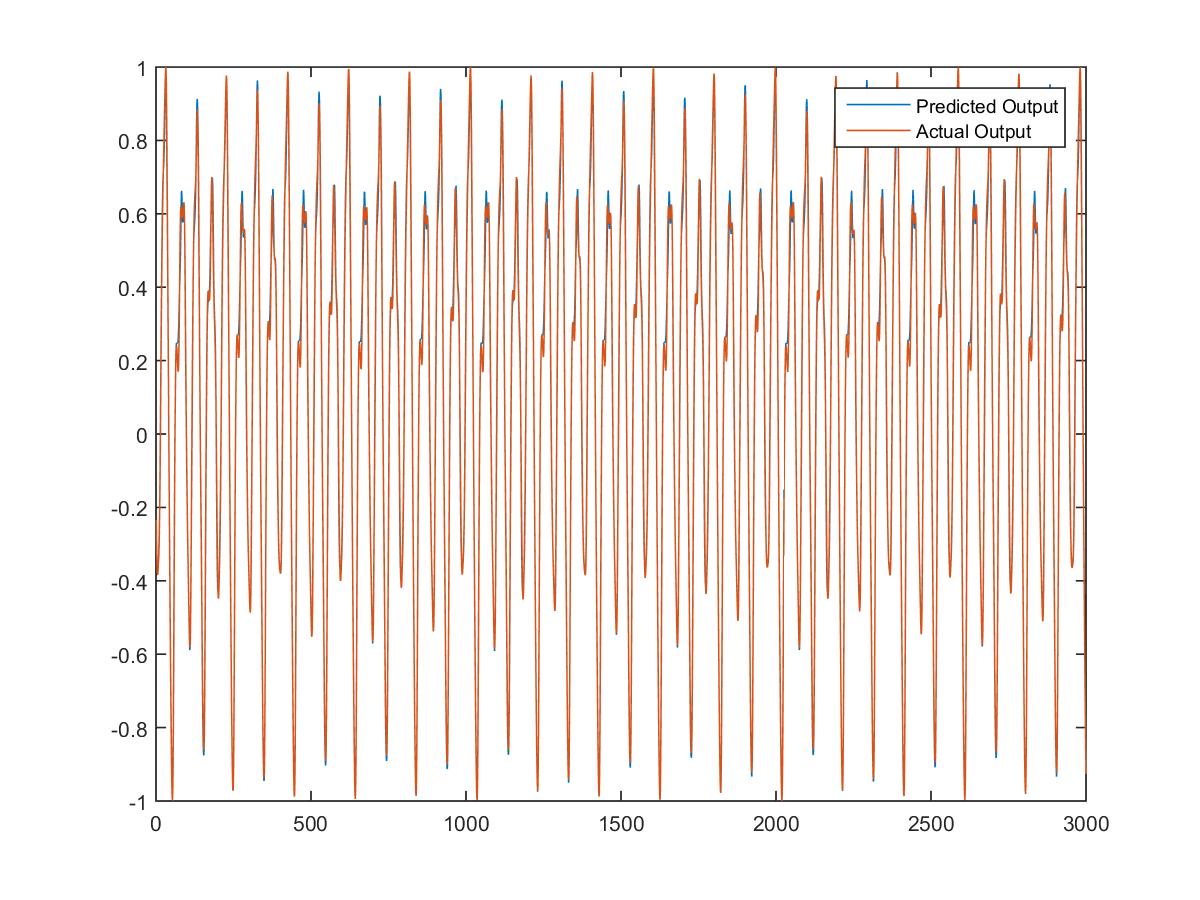
1. Training :



(b) testing:



For problem MG85:

(a) training: 

(b) testing:

