NEURAL NETWORKS
Assignment 1&2 (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 (Multi Layer Perceptron With with Gradient Descent & LS and Radial Basis Functions with Gradient Descent & LS) and five approaches for classification (Different variants of RBF and MLP).

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:

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:

$$\varphi(v) = \frac{1}{1+\exp(-av)}$$

→ 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 for 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 :

```
if(rem(sa,10)==0)
    Wi = Wi + DWi;
    Wo = Wo + DWo;
    DWi = zeros(hid,inp);
    DWo = zeros(out,hid);
end
```

→ The loss function used is:

Least Squared :

$$Least\ Square\ Error\ = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

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

$$\delta_h \leftarrow o_h(1-o_h)\sum_{k \in outputs} w_{h,k}\delta_k$$

For output layer:

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

And the weights are updated as follows using following equations:

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

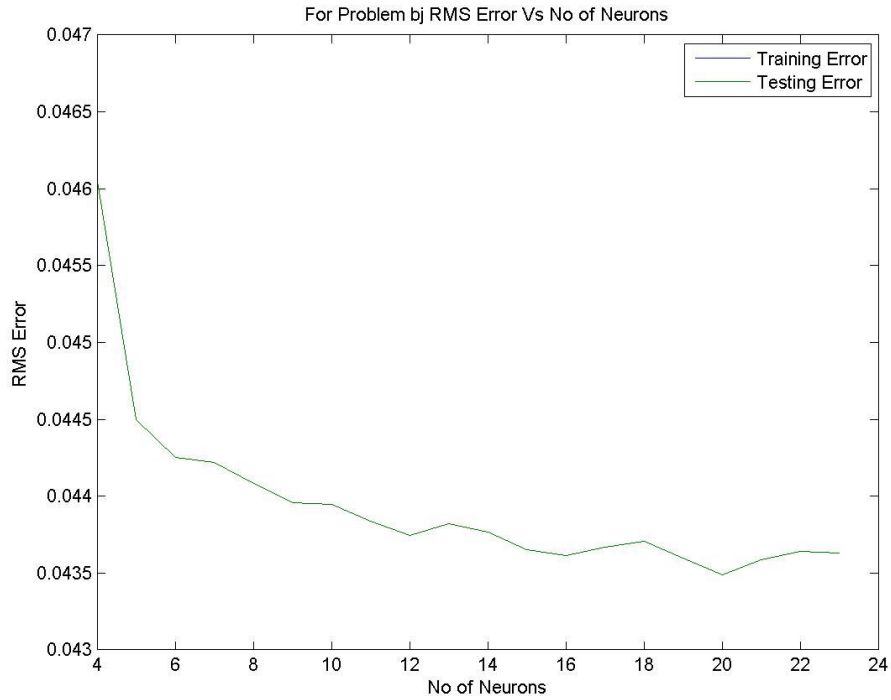
→ After network is trained for training data , the RMSE(Root Mean Square Error)are calculated while validating the training data and during class prediction for the test data.

$$RMSE = \sqrt{\sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{N}}$$

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

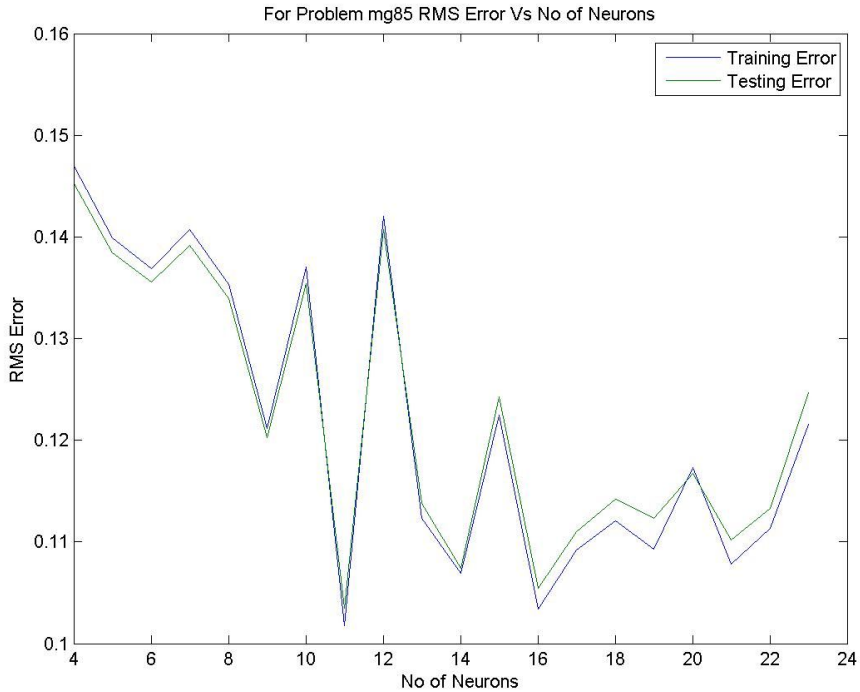
→ Following graph shows how the test and training data average accuracy varies with respect to no of hidden neurons for epoch set to 2000 for Problem BJ with LS loss function. The RMSE not much decreased after hidden neurons = 5. It decreases of order of 0.005's order, which we don't think appropriate 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”

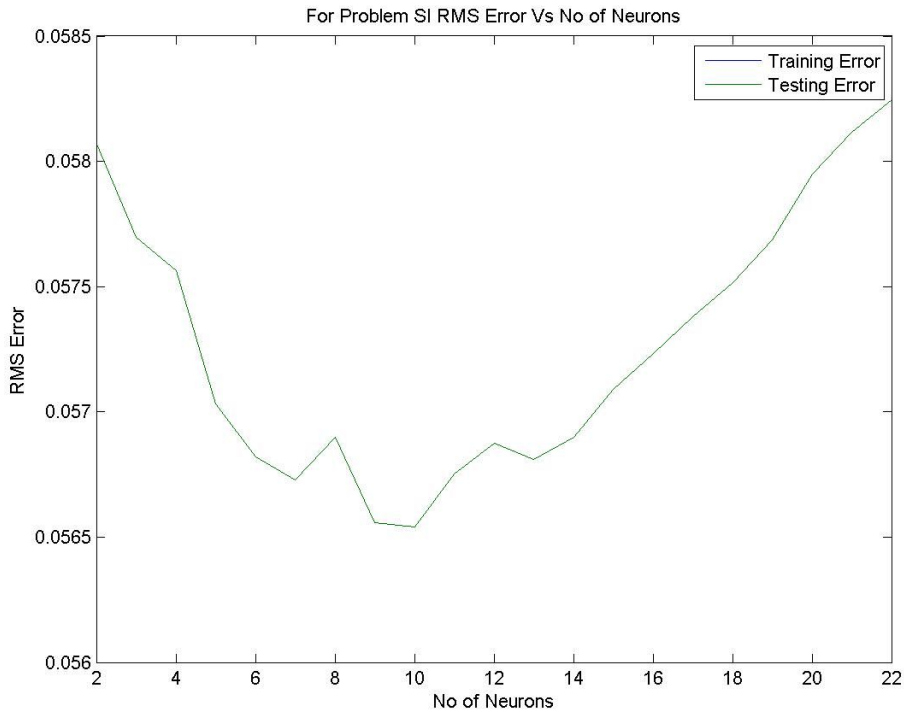


.

(2) For Prob “MG85” :



(3) For Problem SI :



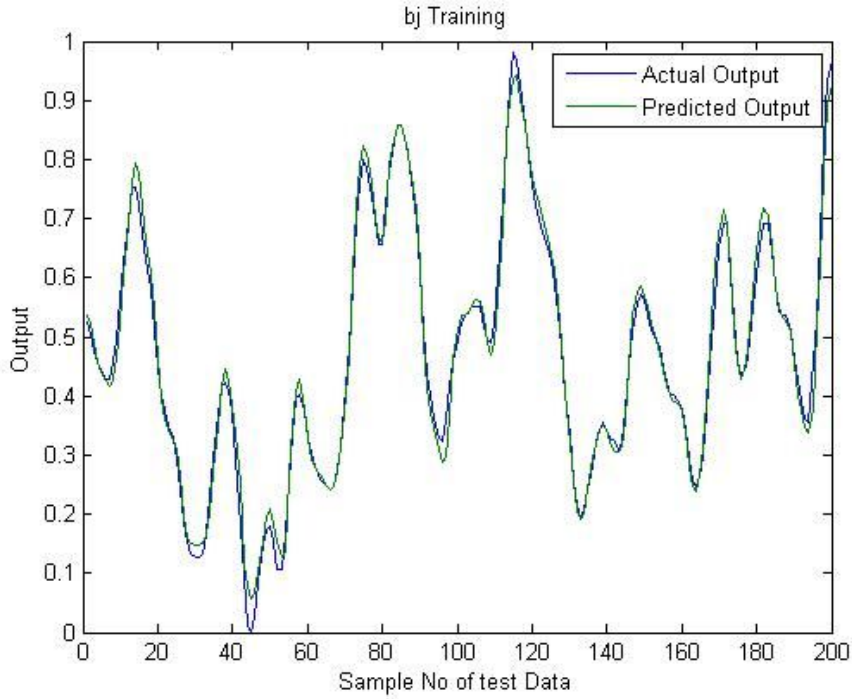
Problem Name	No of Neurons	Epoch	Learning Rate	Training RMSE Error	Testing RMSE Error
BJ	4	2000	0.01	0.023212	0.046148
MG85	11	1200	0.01	0.1017	0.1018
SI	10	20	0.01	0.0563	0.0563

Actual Output Vs Predicted Output Comparison :

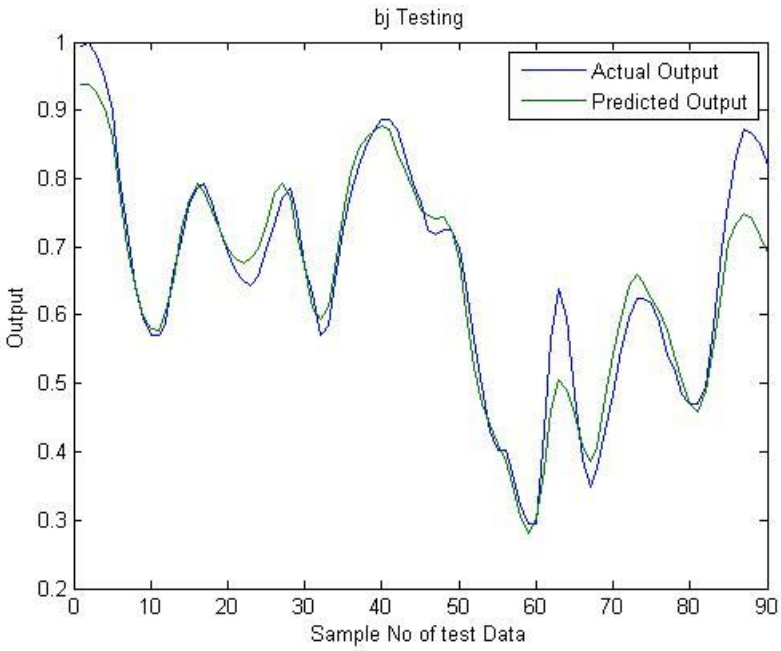
→ Below some more graphs for predicted output and actual outputs comparison for every problem for training and testing.

1. For Problem BJ :

(a) Training :

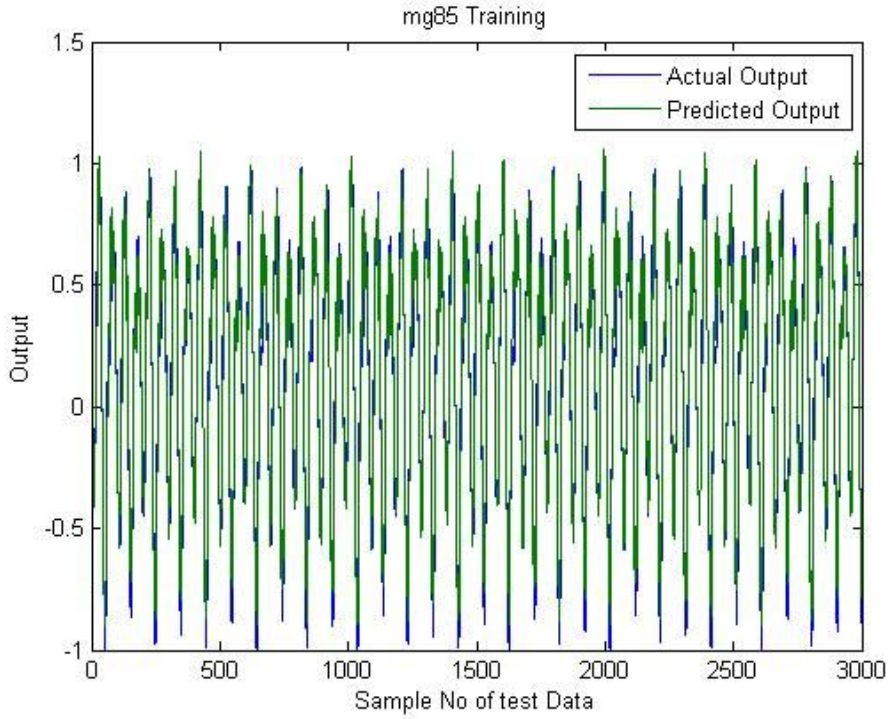


(b) Testing :

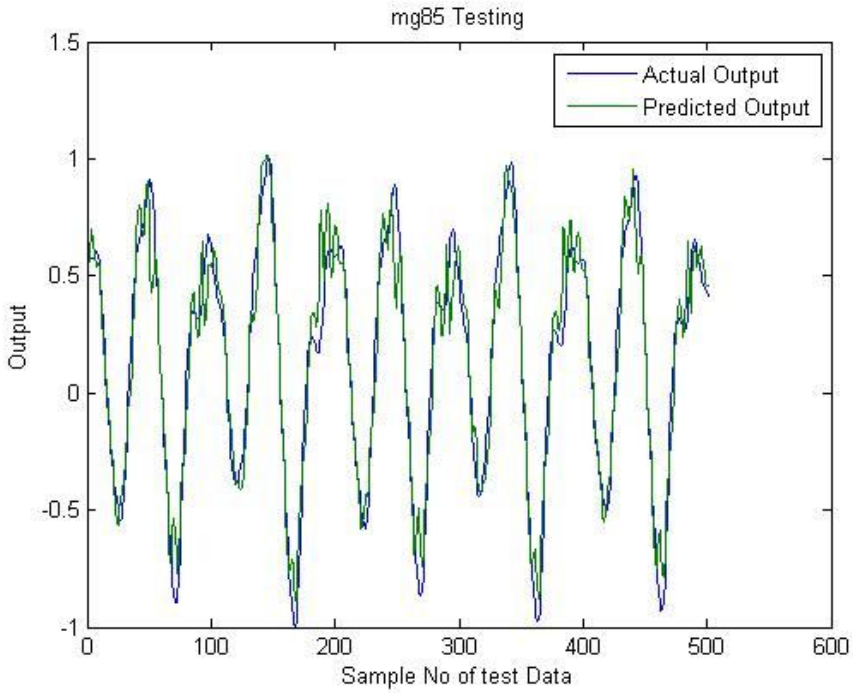


2. For Problem MG85 :

(a) Training :

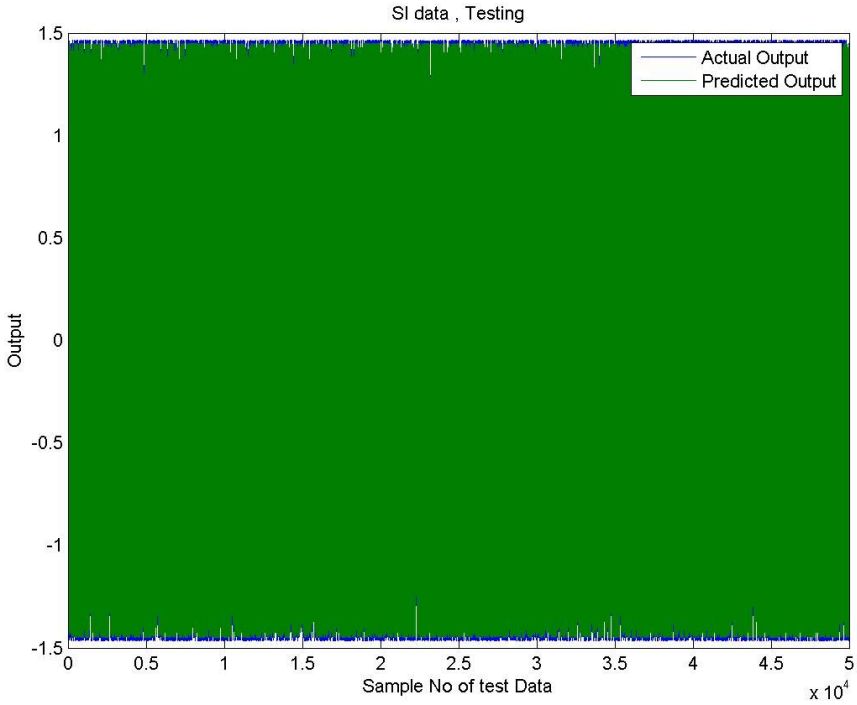


(b) Testing :

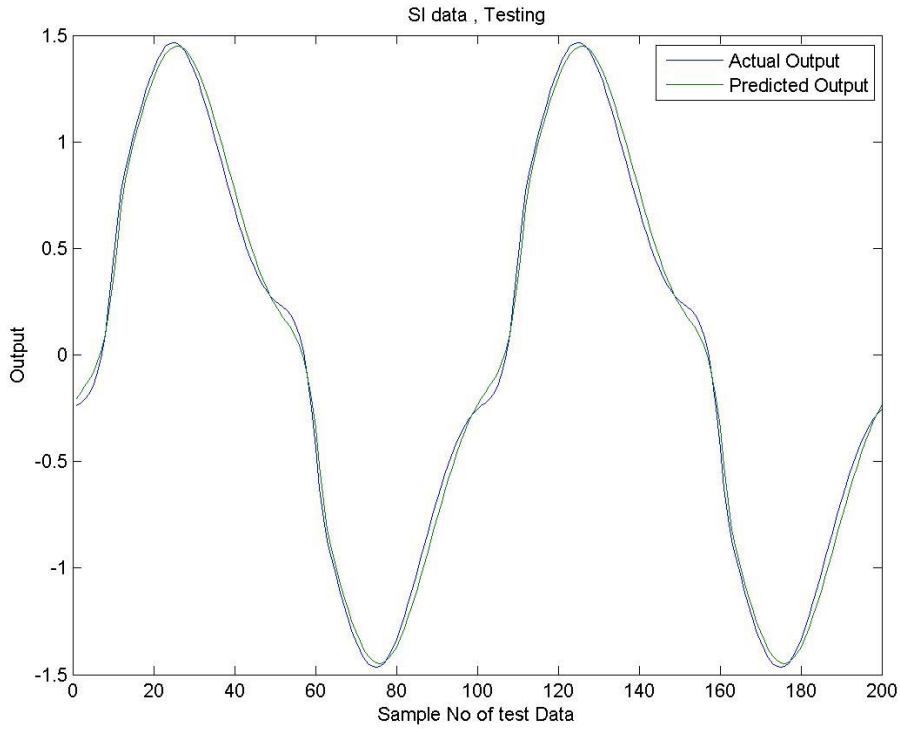


3. For Problem SI :

(a) Training :



(b) Testing :



(2) RBF Approximation:

- *Tanmay Patel , Mit Naria*

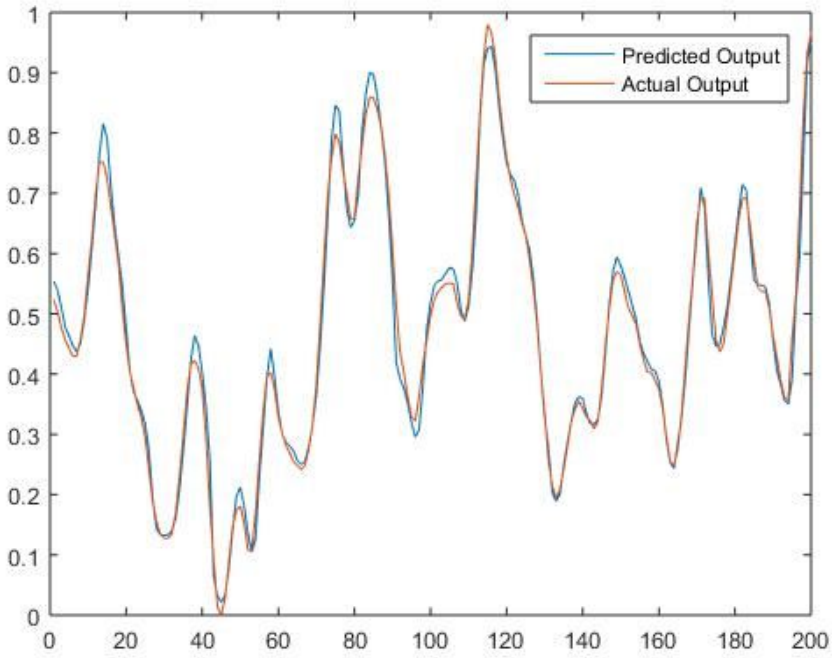
In RBF, radial basis functions are used as activation functions in neurons. First we initialize neurons with random data points selected from input and we also initialize spreads with near zero random values. Then , we train network by feeding it data and then optimizing spreads, weights and sigmas through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

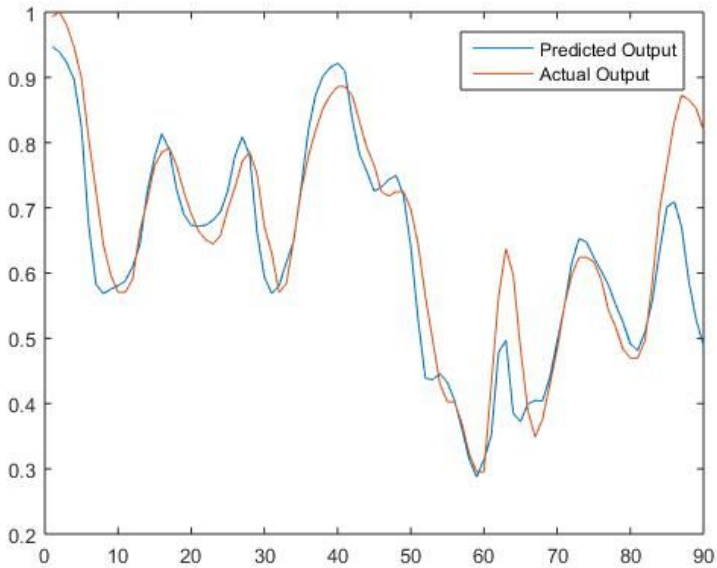
Actual Output Vs Predicted Output Comparison :

For Problem BJ:

(a) training : Function value vs sample no:

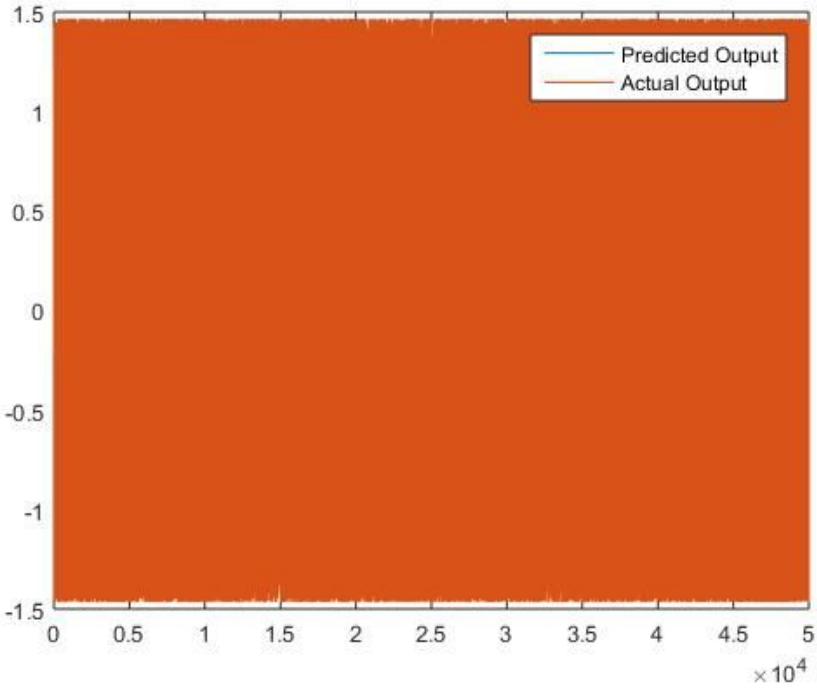


(b) testing : function value vs sample no:

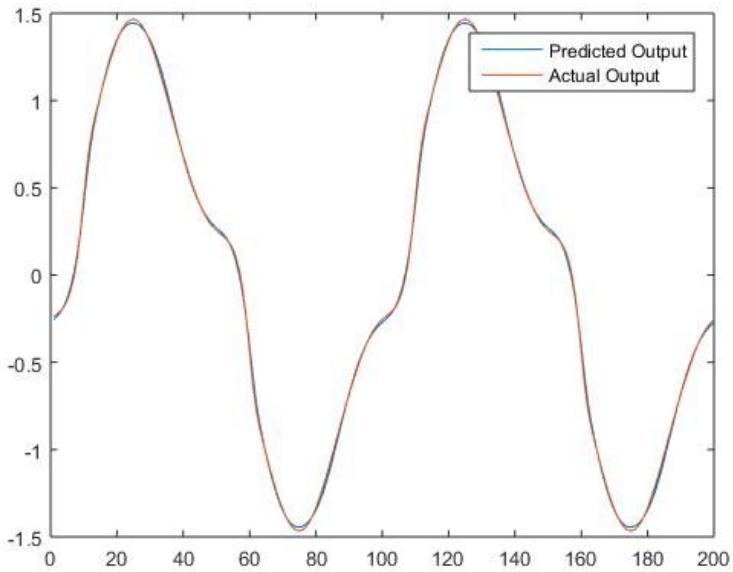


For Problem SI:

(a) Training :

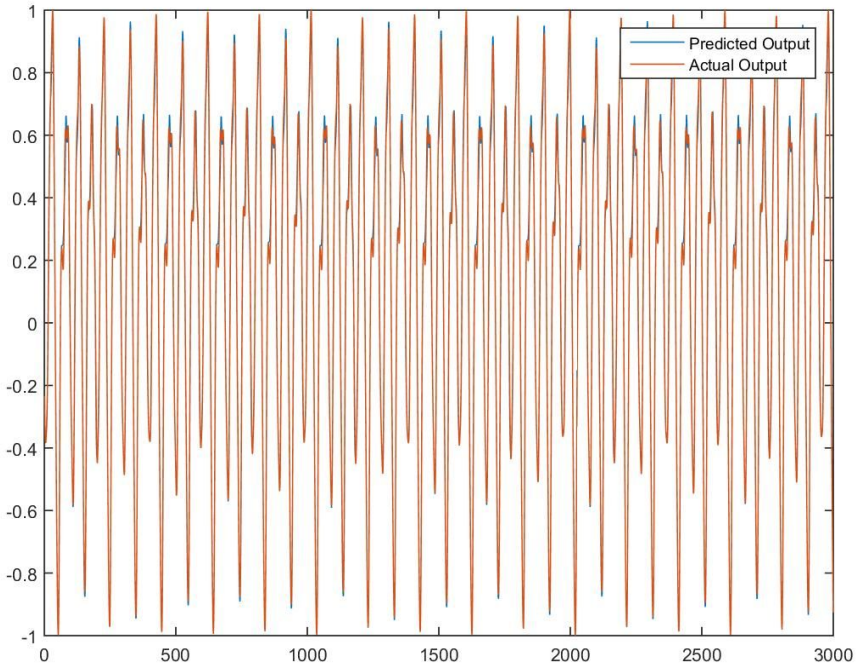


(b) testing:

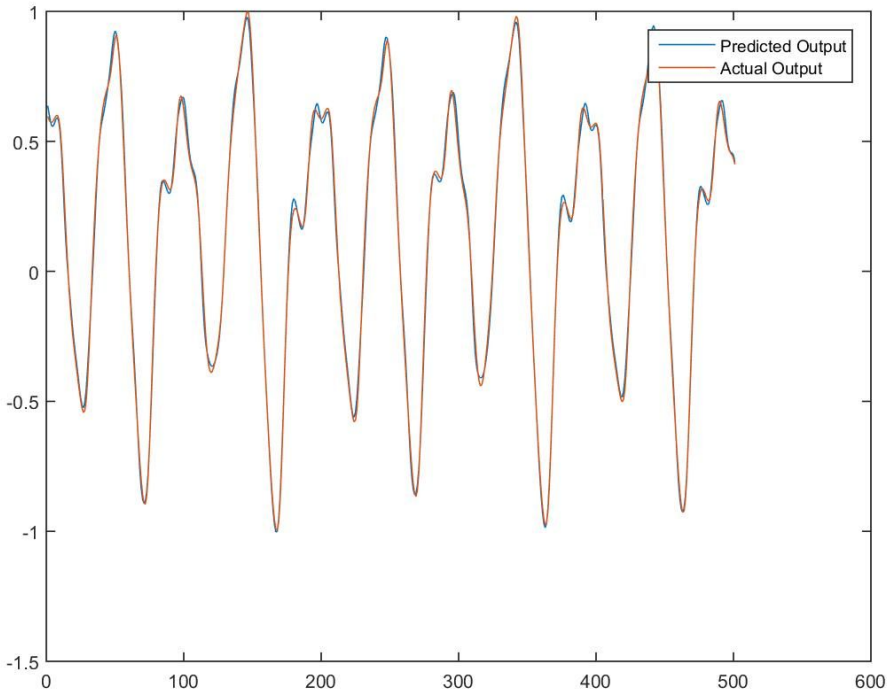


For problem MG85:

(a) training:



(b) testing:



CLASSIFICATION

The patterns that we need to classify, have certain features, a combination of which binds them to a particular class. This enables us to use neural networks for pattern classification. In the same way as function approximation, we can train the network using a dataset with all the required features and the class that the instance belongs to. Using this, the network will learn to map the combinations of features to different classes.

MLP

-By Akshay Miterani, Mihir Limbachia

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 the coded class labels with 1 indicating complete presence and -1 indicating absence.However the general outputs tend to between -1 and 1. If greater than 1 in LS, we penalise the weight matrix to get it below 1.Hence the max among the class label vector is taken to get the predicted class.

The activations functions are :

Log sigmoid:

$$\varphi(v) = \frac{1}{1+\exp(-av)}$$

Purelin:

Linear	$a = n$		purelin

The weight matrix for the hidden layer and the output layer of the two layer perceptron used are assigned randomly and 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.

The loss functions used are:

Least Squared :

$$\text{Least Square (LS):}\ \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}(\hat{y}_{ij}-y_{ij})^2$$

Modified Least Squared :

Modified Least Square (MLS): $\begin{cases} 0 & \text{if } y_{ij}\hat{y}_{ij} > 1 \\ \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}(\hat{y}_{ij} - y_{ij})^2 & \text{otherwise} \end{cases}$

The loss function is minimised through gradient descent learning using following equations:
For hidden layer:

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

For output layer:

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

And the weights are updated as follows using following equations:

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

After network is trained for training data , the confusion matrices are calculated while validating the training data and during class prediction for the test data.

The following measures are used for measuring the performance of this neural network.

Overall Accuracy

$$\eta_o = \frac{100}{N_T}\sum_{i=1}^{C} q_{ii}$$

Average Accuracy

$$\eta_a = \frac{100}{C}\sum_{i=1}^{C} q_{ii} / N_i$$

Geometric Mean Accuracy

$$\eta_g = \sqrt[C]{\prod_{i=1}^{C}\left(100 \times q_{ii} / N_i\right)}$$

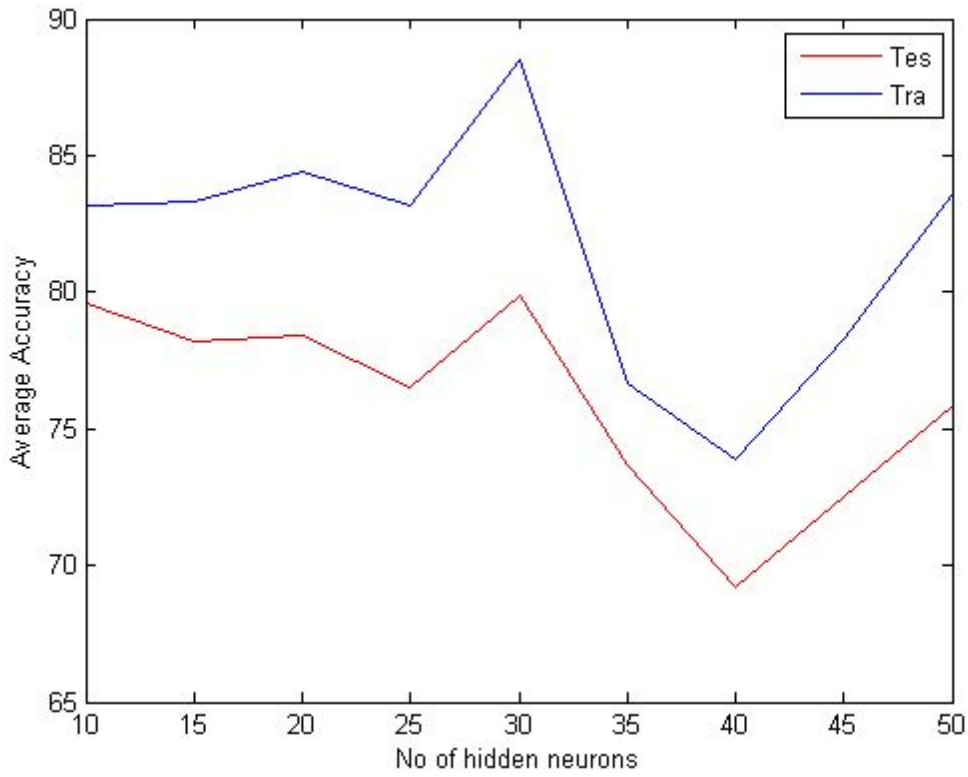
The no of hidden layer neurons and no of epochs are selected taking maximising the average accuracy into consideration.

MLP with Least Square

-By Mihir Limbachia

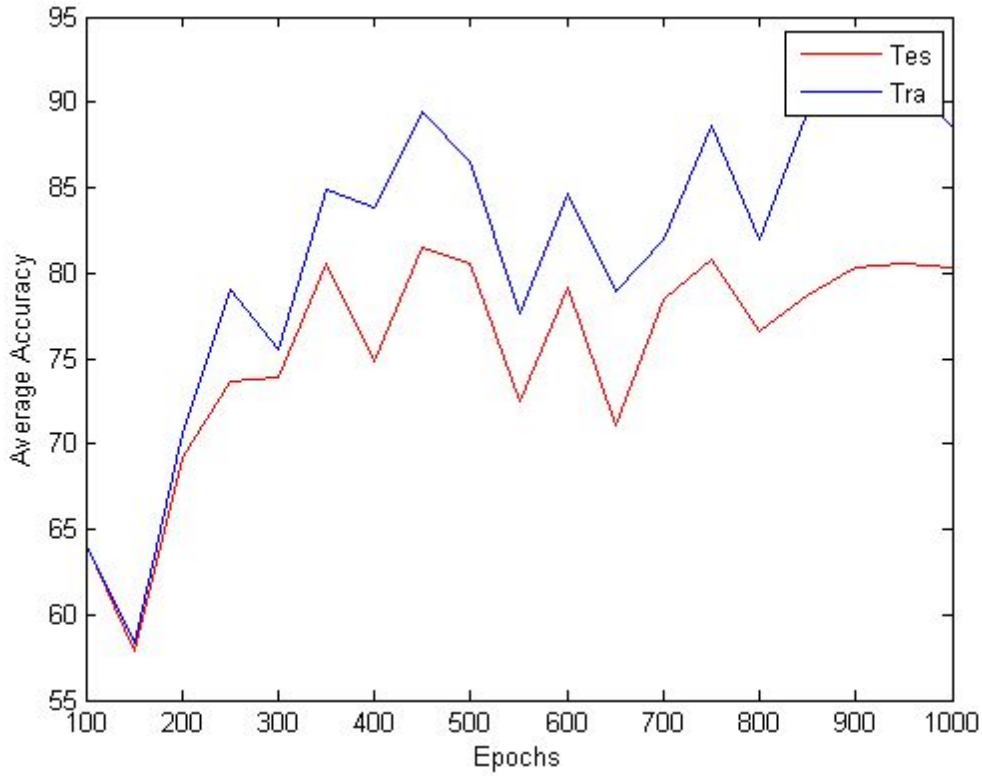
Following graph shows how the test and training data average accuracy varies with respect to no of hidden neurons for epoch set to 750 for VC dataset with LS loss function. The average accuracies is maximised at 30 hidden neurons.

Following graph show



s

that the the maximum accuracy is achieved at 750 epochs so for VC dataset with LS loss function, the number of epochs are set to 750 and hidden neurons to 30. Same procedure is followed for other datasets.



The average results produced for the 10 datasets for MLP classification with LS loss function are as follows:

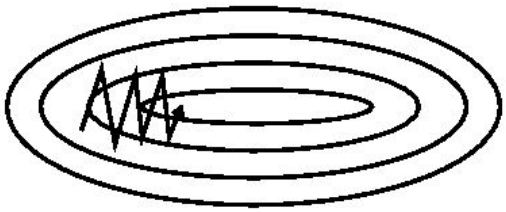
Dataset	**No. of hidden neurons used**	**Overall Accuracy**	**Average Accuracy**	**Geometric Mean Accuracy**
Ae	6	98.2481752	97.78493446	97.71432147
ION	8	82.6902961	78.67991251	75.9225437
Iris	8	95.14285714	95.14285714	94.95750076
Liver	11	69.3826075	69.22300784	68.09786161
PIMA	10	75.73369565	74.61018353	74.70397362
VC	30	76.82699963	67.65068776	76.23150392
Wine	7	95.7627	96.4741	96.5845

MLP with Modified Least Square

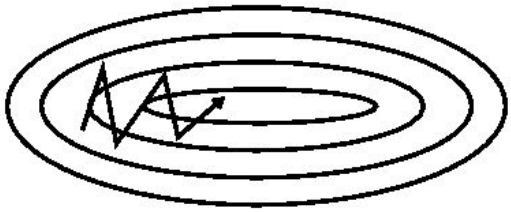
-By Akshay Miterani

Code Name: MLPClassification_MLS_momentum_AkshayMiterani.m
*Extra Efforts: MLP with modified least square **with momentum term***

Explanation:
Stochastic gradient descent tends to oscillate in some cases and hence we need a momentum term to reduce the oscillation and have easy convergence



Stochastic gradient descent



Gradient descent with momentum

Equations used in code

$v_t = \gamma v_{t-1} + \eta \quad _\theta J(\theta)$

$\theta = \theta - v_t$

γ is the momentum term which is generally set to 0.9 or similar values.

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Reference : http://sebastianruder.com/optimizing-gradient-descent/

Code Name: MLPClassification_MLS_MiniBatchUpdate_AkshayMiterani.m
*Extra Efforts: MLP with modified least square with **mini batch update***

We can either update the weights in a batch or sequentially for every weight. However, to think for a hybrid, lets update in batches of finite size. So for instance, we divide the Wine class into 3 batches and store their update and then use batch

update to update the weights for the current mini-batch and again initialize accumulations to zero.

Code Snippet:

```
b = mod(int32(iteration_number) ,int32(Mini_Batch_Size));
        if(b==0)
          Wi = Wi + DWi;
          Wo = Wo + DWo;
          DWi = zeros(hid,inp);
          DWo = zeros(out,hid);
        end
```

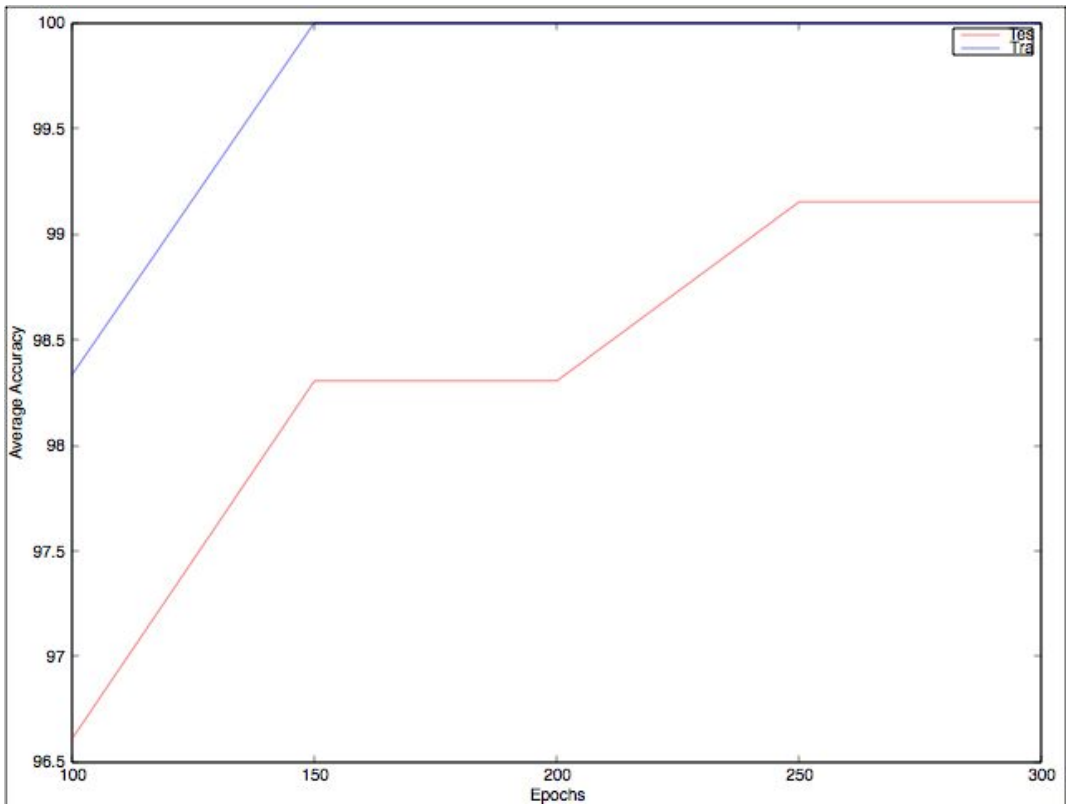
Now the challenge is how many mini-batches to create.

Analysis for Stochastic gradient descent

Code Name: MLP_MLS_BatchUpdate_AkshayMiterani.m

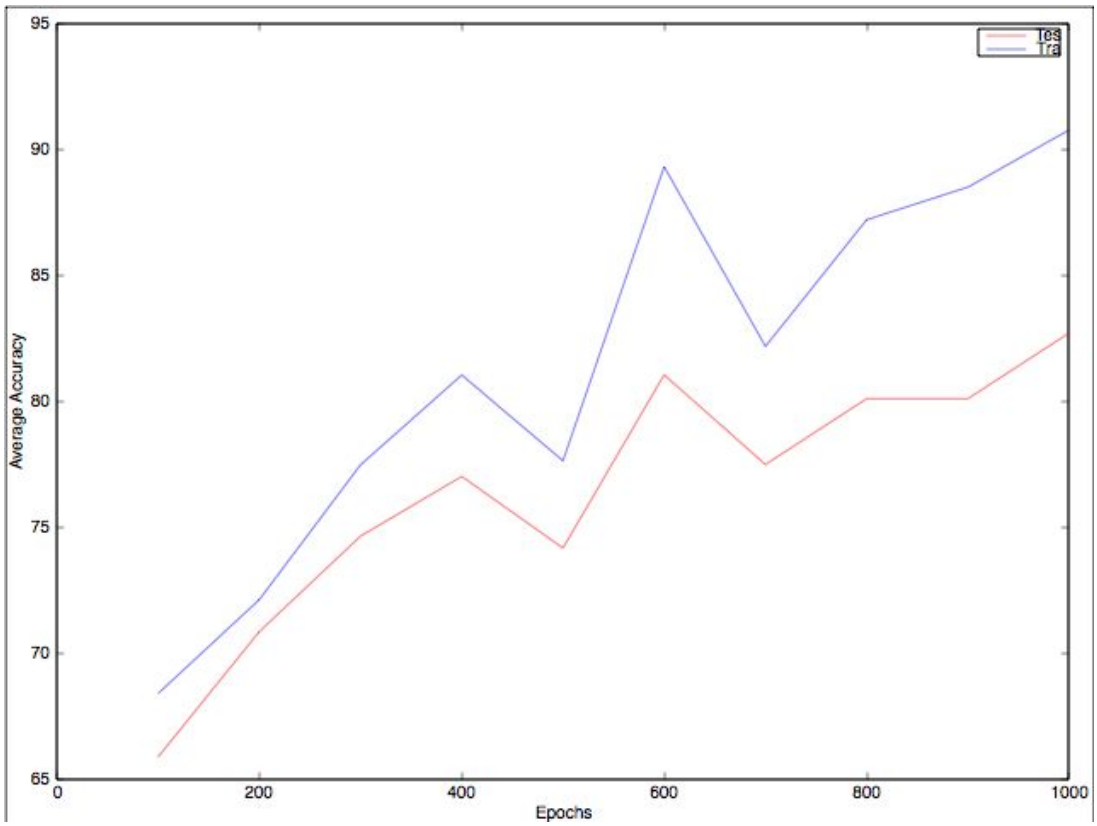
For Wine data set when we run a loop on number of epochs for optimal number of hidden neurons and we get the following graphs

MLP_MLS_WINE

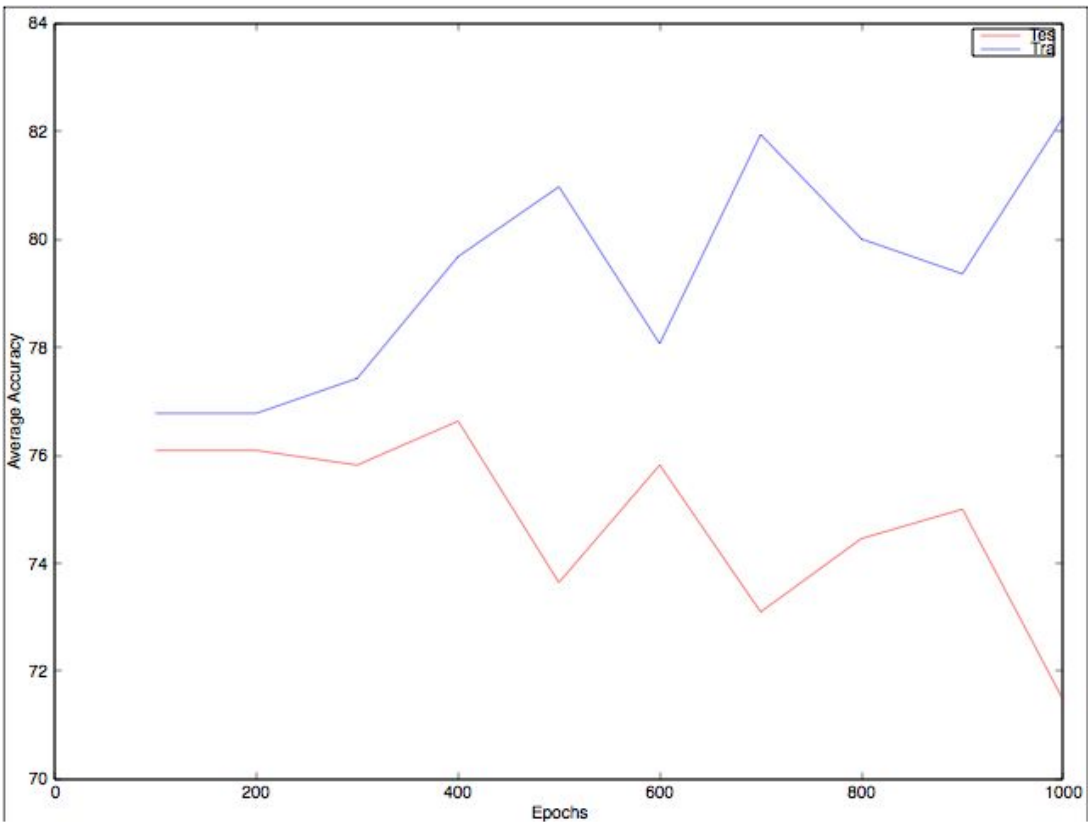


We follow the same approach for all the data sets and plot the graphs.

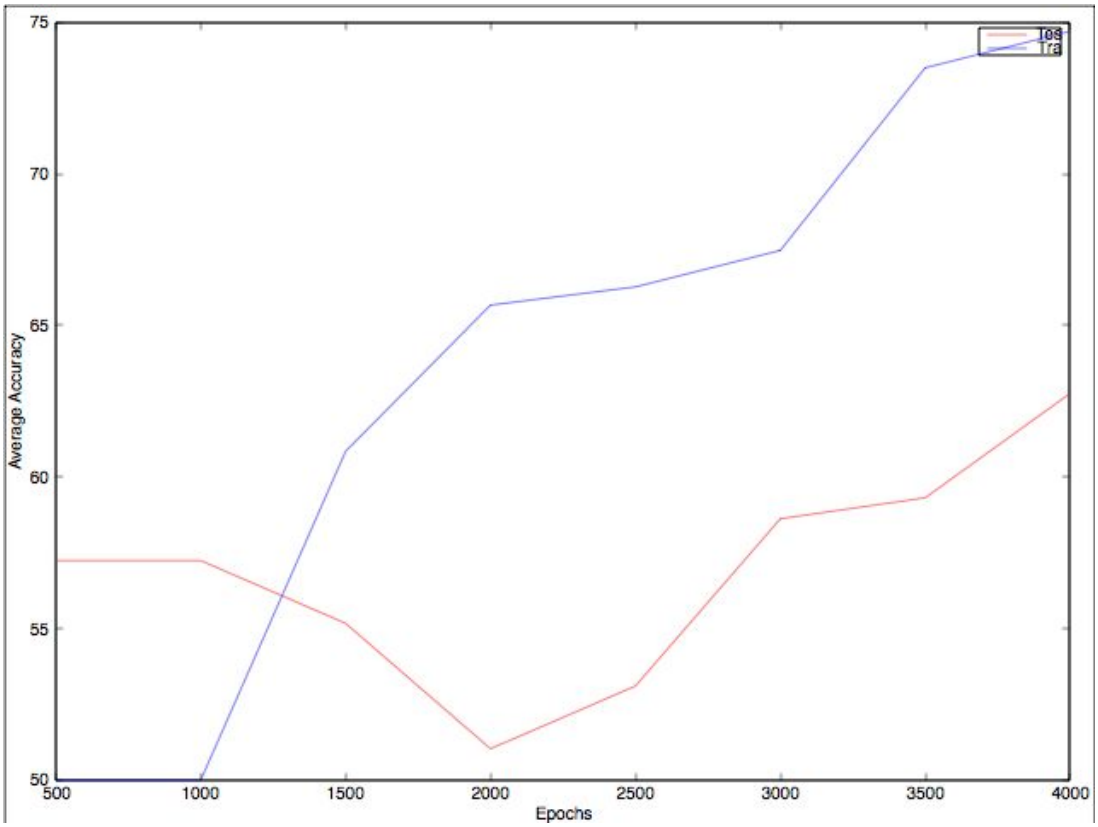
MLP_MLS_VC



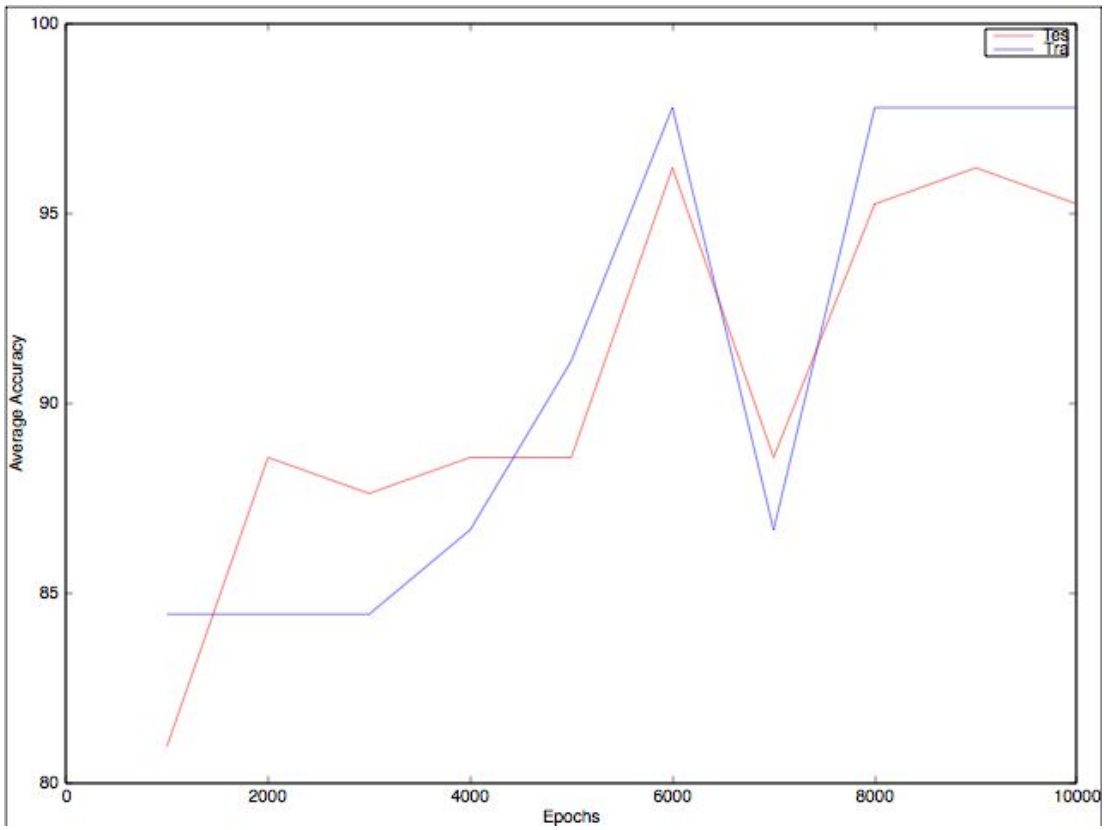
MLP_MLS_PIMA



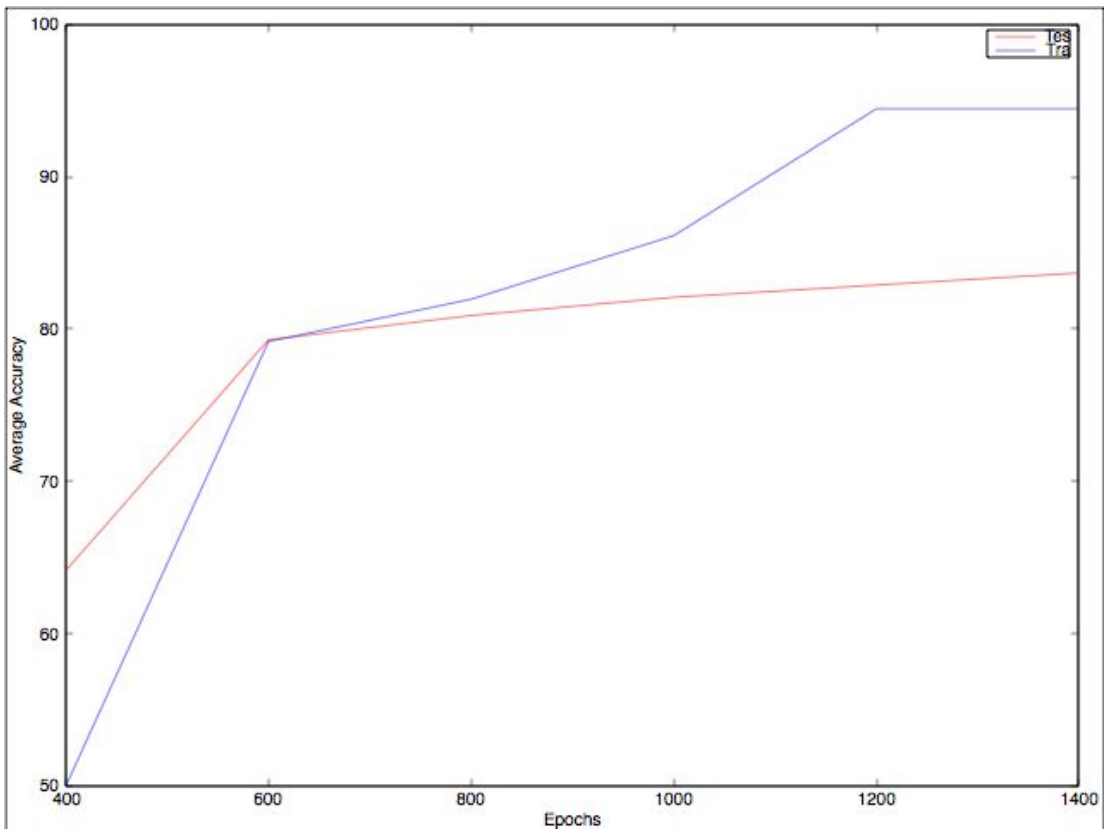
MLP_MLS_Liver



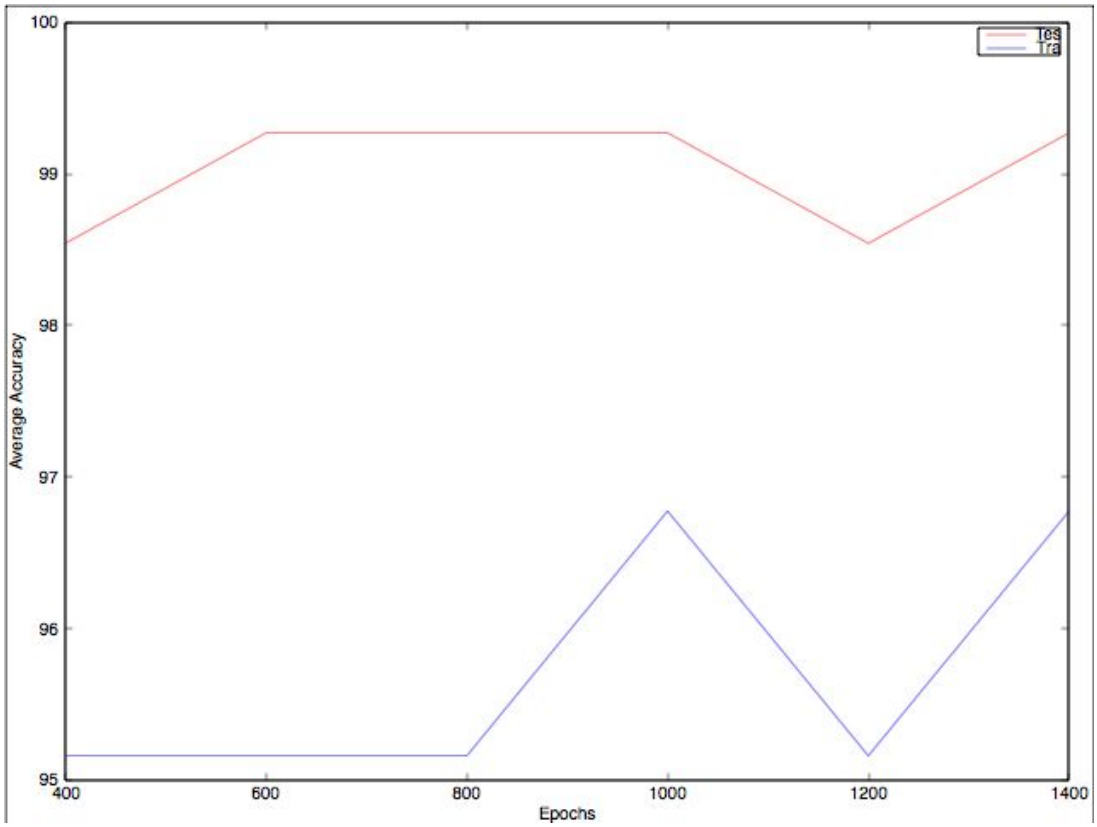
MLP_MLS_Iris



MLP_MLS_ION



MLP_MLS_ae



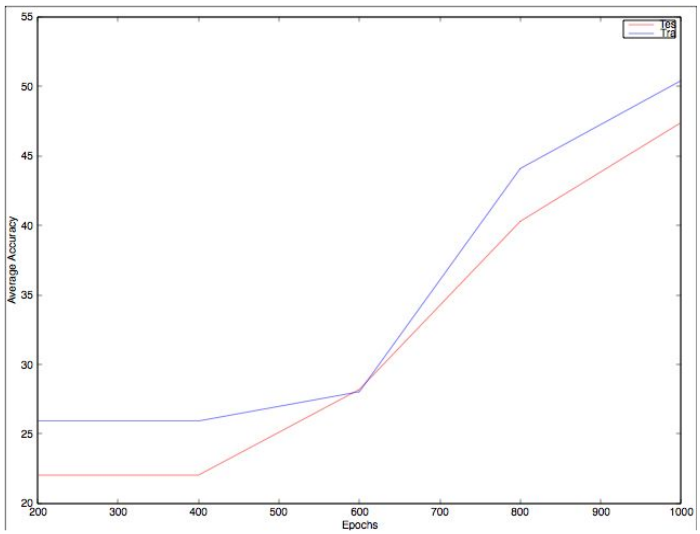
So now, the task was to select the optimal rate which suffices our need, keeping the number of neurons low and number of epochs less. Hence we found out the optimal number of neurons and epochs which are mentioned in the table below.

Interesting observation is when we compare MLP with least square and MLP with modified least square we get better results in the later one. In most of the data sets we see that training accuracy keeps increasing with number of epochs and the test

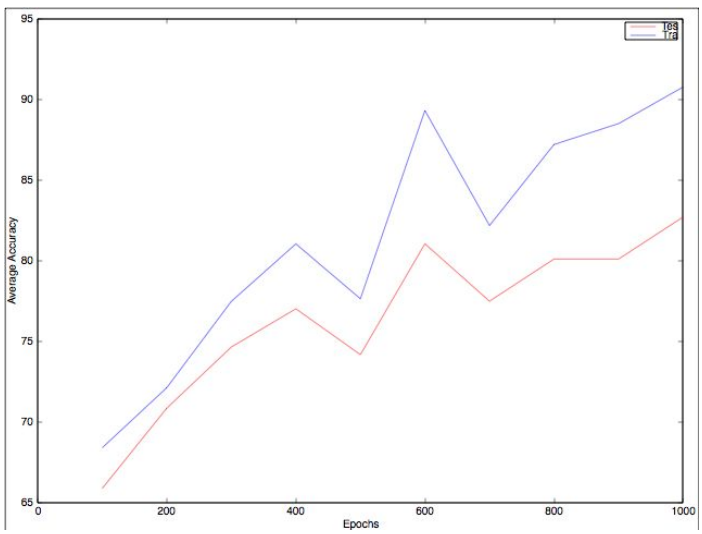
Also in the above graphs we are able to see that when we increase the number of epochs we see that the training data reaches maximum accuracy. However, this is not a good sign as this shows that our model has now started mimicking the data which will give good results mostly for items from the training set. This is known as over training the model.

For dataset ION and VC, we tried sequential update as will as batch update. We found the following results.

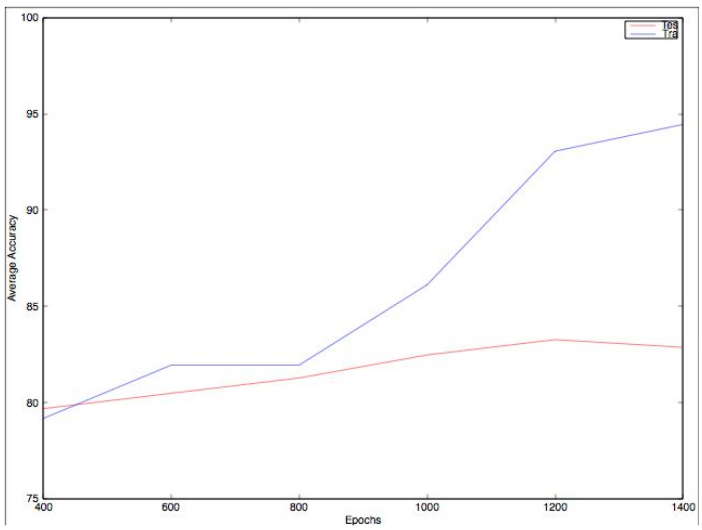
VC_MLP_MLS_Batch_Update



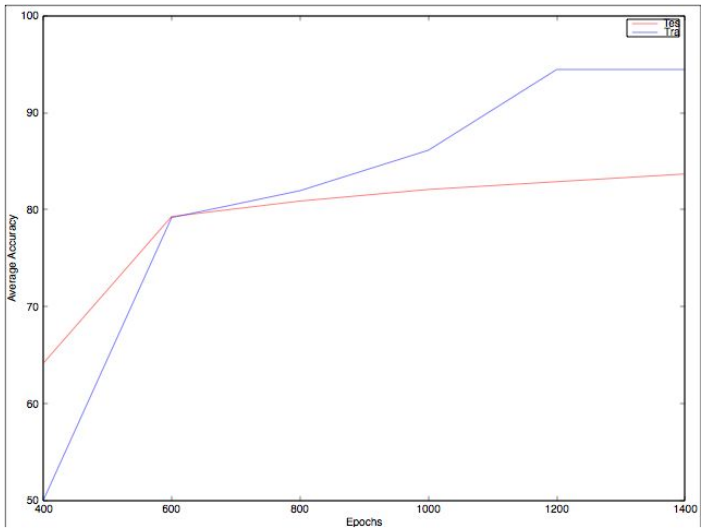
VC_MLP_MLS_Sequential_Update



ION_MLP_MLS_Batch_Update



ION_MLP_MLS_Sequential_Update



Hence we see that for less number of epochs the sequential update has less accuracy which on introspection turns true. Say for case where the data is highly random, when we do sequential update the weights shift too much even though they should not. So after checking for different sets, we have decided to use different techniques as far as MLS classification is concerned.

For MLP with Modified Least Square in accordance to graphs above :

Dataset	**No. of hidden neurons used**	**Epochs**	**Overall Accuracy**	**Average Accuracy**	**Geometric Mean Accuracy**
Ae	6	1000	99.270	99.156	99.167
ION	8	1000	85.259	80.646	81.649
Iris	8	8000	96.190	96.124	96.190
Liver	11	3000	70.345	70.422	70.424
PIMA	10	800	75.543	74.143	74.222
VC	40	750	76.066	73.924	76.275
Wine	7	300	98.305	98.145	98.156

RBF

- *Mit Naria, Urmil Kadakia, Tanmay Patel, Deep Raiya, Anusha Phadnis*

In random initialization approach , we randomly select data points from training data and take them as centers of Radial Basis Functions. We calculate spreads from these selected data using formula:

$$\sigma = \frac{\text{Maximum distance between any 2 centers}}{\sqrt{\text{number of centers}}} = \frac{d_{max}}{\sqrt{K}}$$

Other method is using K-means clustering to find K centers of Radial Basis Function.

RBF Random Initialization with Pseudo Inverse

-By Tanmay Patel

In this approach , we obtain weights by solving equation:

$$G\begin{bmatrix} w_1 \\ \dots \\ w_K \end{bmatrix} = \begin{bmatrix} d_1 \\ \dots \\ d_N \end{bmatrix}$$

where d_i is coded class label for class i and G is given by below equation:

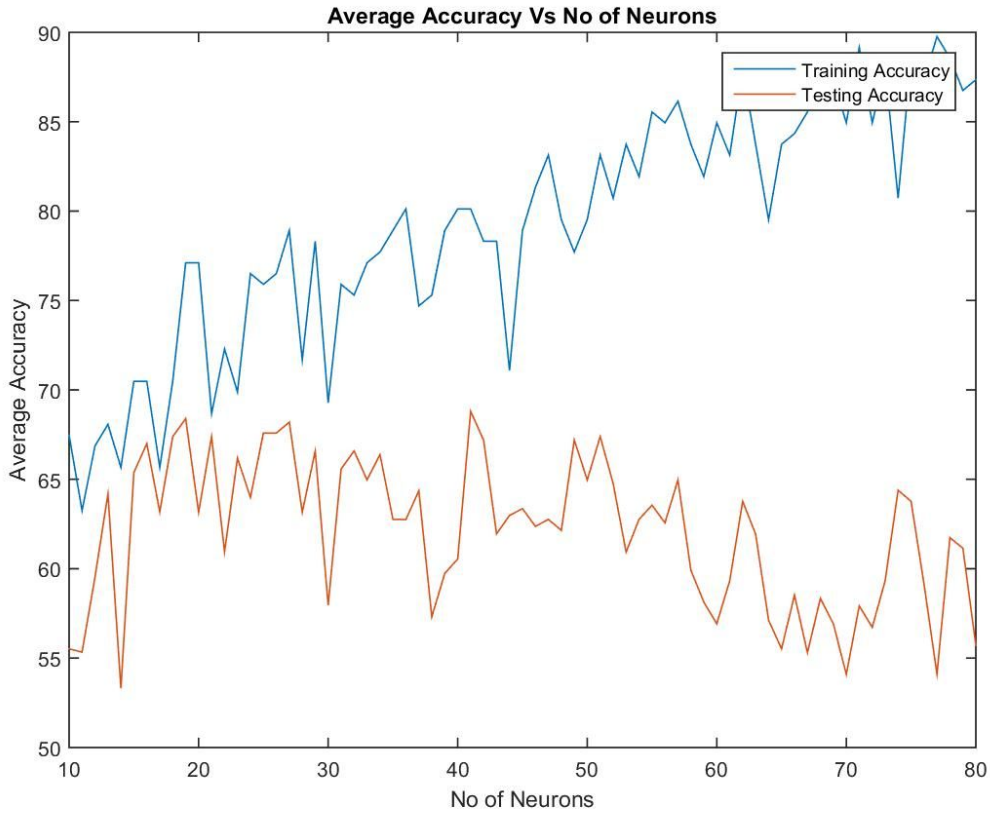
$$G = \begin{bmatrix} \phi_1(\| x_1 - \mu_1 \|) & \dots & \phi_K(\| x_N - \mu_K \|) \\ \vdots & \dots & \vdots \\ \phi_1(\| x_N - \mu_1 \|) & \dots & \phi_K(\| x_N - t_K \|) \end{bmatrix}$$

where ϕ_i () is radial basis function of hidden neuron i with center μ_1 .

In this method weights are found by taking pseudo inverse of matrix G and multiplying it with the expected output. Hence it does not need more than one run(epoch) for training. However, further weights can be optimized using gradient descent.

Following is the graph for average accuracy vs number of neurons for Liver dataset. Training accuracy is increasing curve because with increasing number of neurons training samples are predicted more accurately as the distance of samples from centre of hidden neurons decreases. But the testing accuracy does not have the same trend, because overtraining may decrease the testing accuracy. The optimum

number of neurons in this case is 16. Similarly plotting graph for other dataset will give an idea about optimum number of neurons.



Dataset	No. of hidden Neurons	Overall Accuracy	Geometric Mean Accuracy	Average Accuracy
AE	6	96.9343	95.8923	96.8723
ION	40	81.3549	76.9023	77.8134
Iris	8	96.001	95.789	95.881
Liver	16	64.6206	59.3188	61.0302
PIMA	12	71.1957	71.5705	71.5773
VC	41	68.7204	65.1410	69.1187
Wine	11	92.3729	92.2351	92.3059

RBF Random Initialization with Gradient Descent and LS :

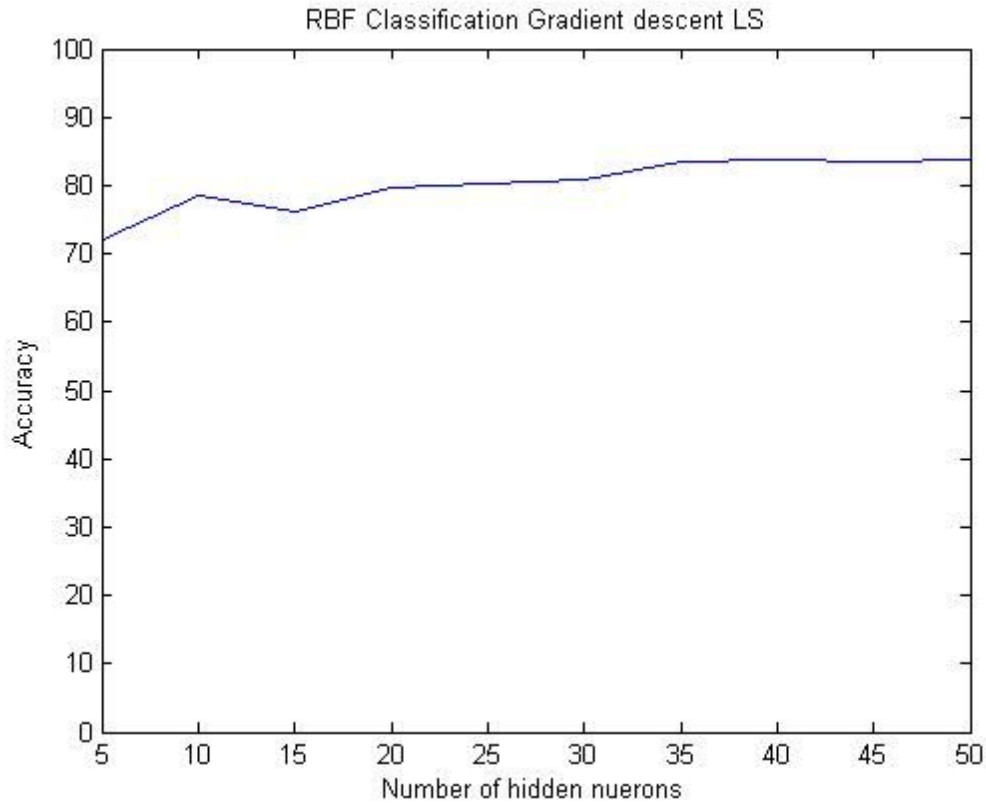
-By Urmil Kadakia

In this approach , we apply supervised learning in order to get good weights. We first initialise weights by random small numbers (~ 0) and then we update these weights as learning process continues. Usually, we try to update weights based on cost function. (We try to update weights for which we get less cost.)

For set 1 dataset VC.

Dataset	**No. of hidden neurons used**	**Overall Accuracy**	**Average Accuracy**	**Geometric Mean Accuracy**
Ae	6	99.210031	99.754623	99.124349
ION	43	87.250996	87.120963	87.122153
Iris	8	97.142857	97.114838	97.142857
Liver	16	66.206897	64.411594	64.972794
PIMA	12	75.000000	73.210030	73.336804
VC	41	83.175355	81.180215	82.828245
Wine	11	97.457627	96.718919	96.764347

Here just I have shown one graph For Problem VC set 1 for finding hidden neurons(for epoch =750(fixed)) and as I pointed out in graph for hidden neurons = 41, we get maximum Geometric Mean Accuracy which is around 82%.



RBF (K means clustering with Pseudo inverse)

-By Anusha Phadnis

In this approach, instead of initialising the centers of the activation functions randomly, we used K means clustering to find out the centers more accurately, k being equal to the number of hidden neurons in the hidden layer. Clustering gives a better distribution of the centers.

The clustering algorithm works in the following way:

- First, random points from the dataset are taken as centroids.
- Then, the remaining data points are assigned to a particular cluster based on minimum distance.
- Then, if needed, the centroids are re assigned based on the clusters created. This process is repeated until there is no noticeable change in the centroids of the clusters.

The distances, or the spreads of the centers obtained from K means clustering, are calculated in a normalized way, i.e. (d_{max} / sqrt(k)).

Once the centers and their spreads are calculated, we get a matrix using Gaussian function as activation function, i.e.

$$\phi_i = \exp\left(-\frac{\|x-\mu_i\|^2}{2\sigma_i^2}\right)$$

The matrix obtained and its relation with the second layer weights can be described as follows:

$$\begin{bmatrix} \phi_1(\|x_1-\mu_1\|)\ldots\phi_K(\|x_1-\mu_K\|) \\ \ldots \\ \phi_1(\|x_N-\mu_1\|)\ldots\phi_K(\|x_N-\mu_K\|) \end{bmatrix} [w_1\ldots w_K]^T = [d_1\ldots d_N]^T$$

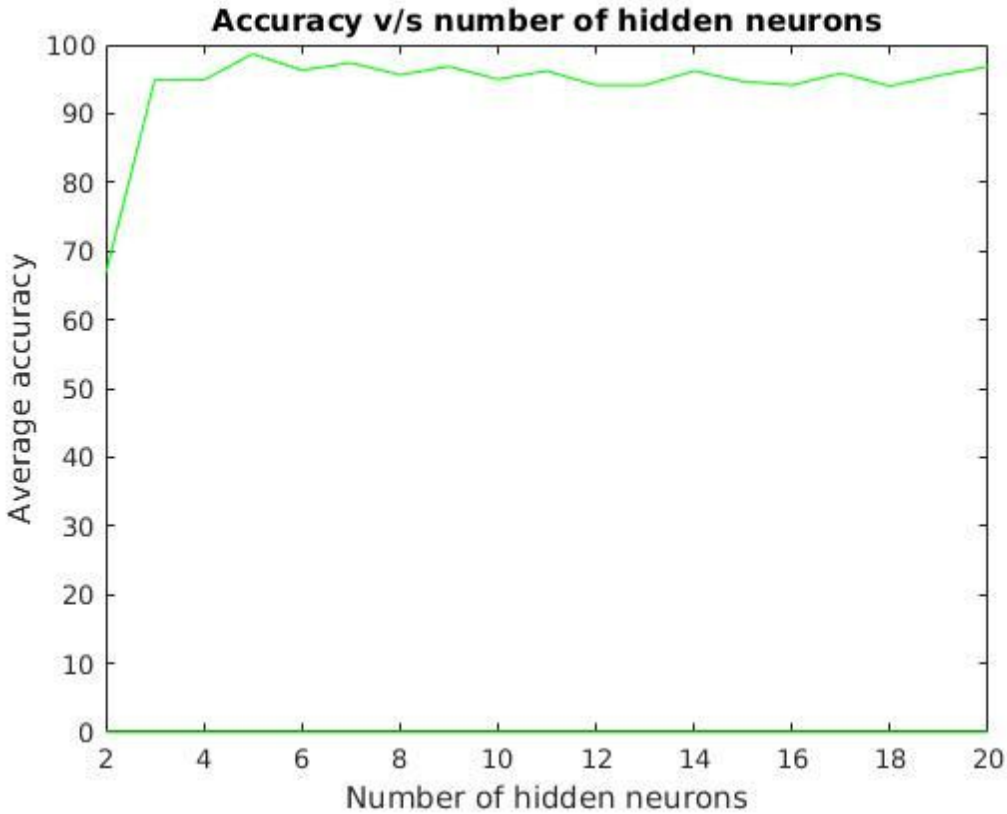
So, to obtain the weights required, we calculate the pseudo inverse of the above matrix and multiply it with the output values from the training dataset. The least square error loss function also converges to using the pseudo inverse matrix for this function.

While testing data, the matrix with Gaussian function is calculated again based on clustered centers and distances, and then multiplied with the weights to obtain the output. The average results obtained with the given testing datasets were as follows:

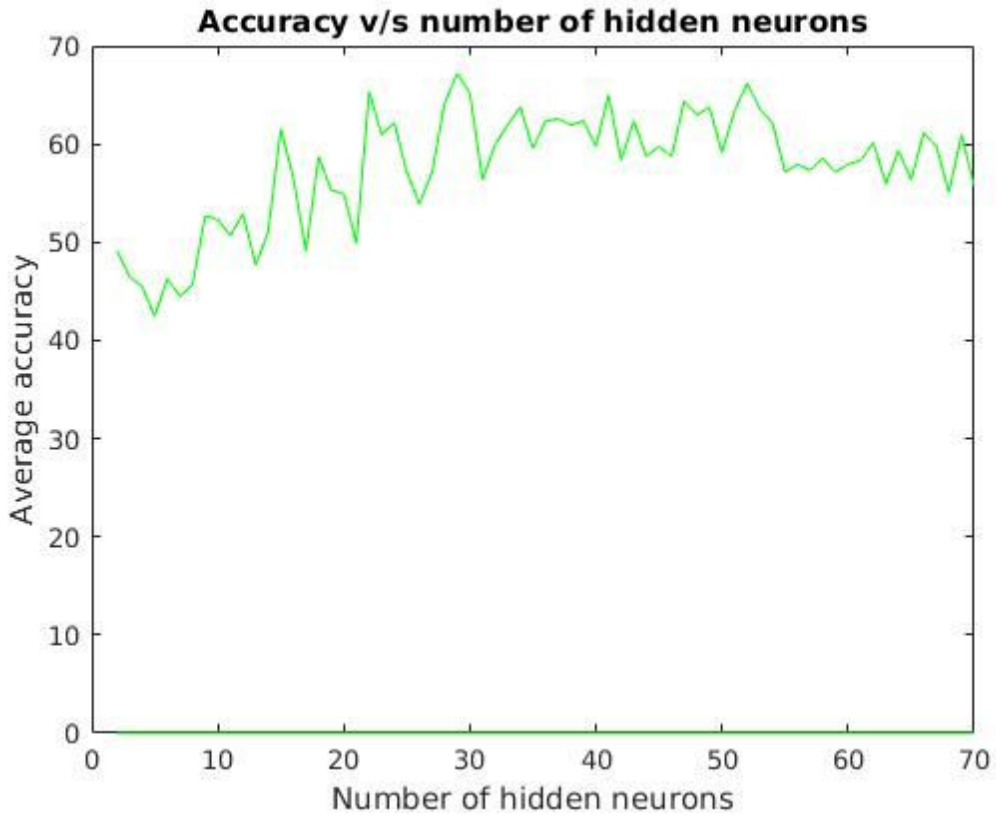
Dataset	**No. of hidden neurons used**	**Overall Accuracy**	**Average Accuracy**	**Geometric Mean Accuracy**
Ae	6	99.09349	98.00754	98.4177167
ION	15	82.23107	78.92166	77.87952
Iris	8	97.04763	97.04763	96.98449
Liver	30	66.48277	66.35638	66.28419
PIMA	8	72.20109	73.87974	73.60322
VC	50	72.51185	72.6245	69.76497
Wine	6	95.42375	96.07482	95.98766

While using this approach, the only parameter that can be varied is the number of hidden neurons, or the number of cluster centroids formed. In most cases, the accuracy initially increases as the number of hidden neurons increases, until a certain point, where it reaches the maximum accuracy possible. This increase in accuracy with the number of hidden neurons can be attributed to the fact that more efficient training takes place when there are more neurons, because a single input would go through more number of radially based functions before moving on to the next layer. The accuracy will decrease after a certain number of neurons, due to

overtraining, as the network will start mimicking the training data. As an example, the average accuracy for “Wine” dataset varies with the number of hidden neurons as shown by the following graph:



Since the datasets are varied, the optimum number of hidden neurons to be used cannot be determined functionally. A good way to find out this number would be to plot the accuracy of the testing data against the number of hidden neurons, so that the same number can be used for unknown datasets, since in real world applications, we wouldn’t know the results that we are supposed to obtain. For example, for “Liver” dataset, the graph is as follows:



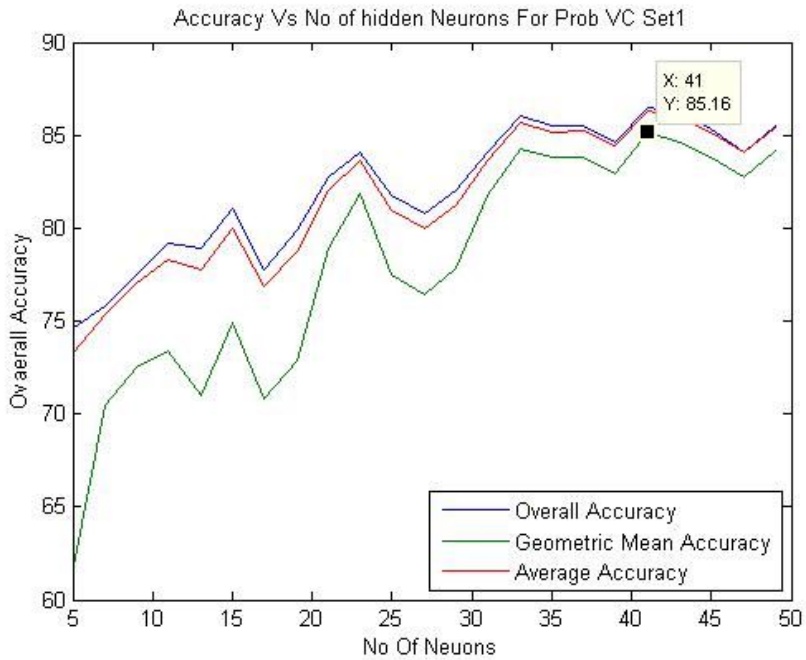
This shows that 30 would be the optimal number of hidden neurons to use while classification of the Liver dataset, which is in accordance with the table of data.

RBF Random Initialization MLS Gradient

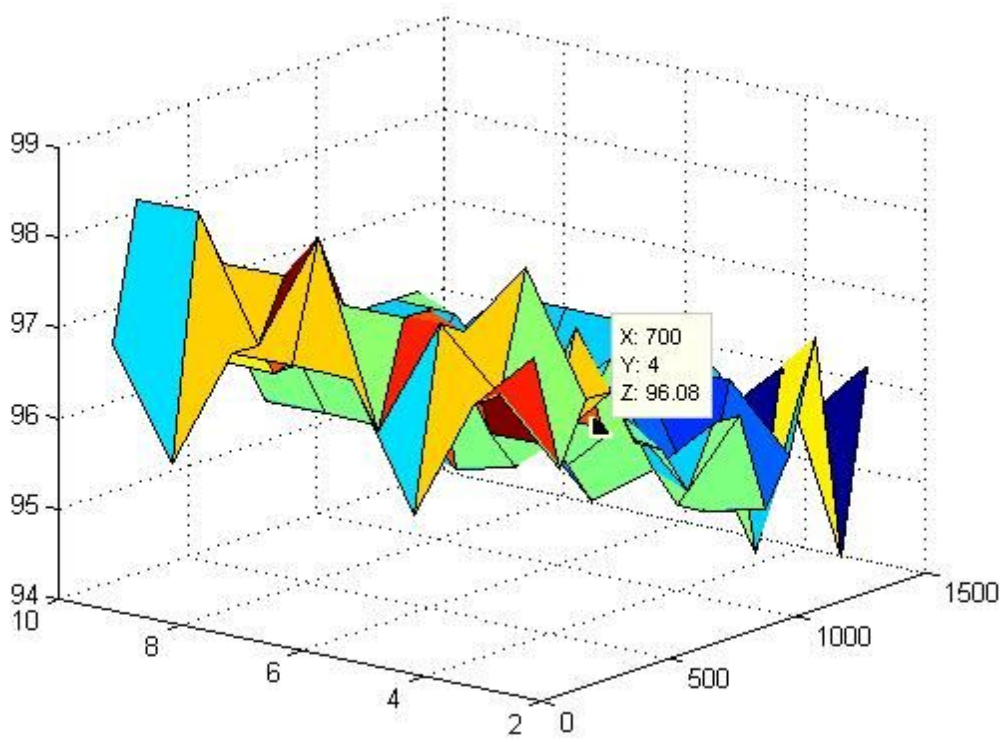
-By Deep Raiya

→ As mentioned above, I have also used same approach for finding hidden neurons and epoch for every problem.

→ Here just I have shown one graph For Problem VC set 1 for finding hidden neurons(for epoch =750(fixed)) and as I pointed out in graph for hidden neurons = 41, we get maximum Geometric Mean Accuracy which is around 85%.



→ For some problem I could not guess epochs and no of neurons so I have done double loop for problem like wine and plot 3d graph and find optimum point.
As you can see we get 96 percentage geometric mean Accuracy for Problem Wine Set 1 is around hidden neurons 4 and 700 epochs.



→ Average accuracy over all 10 sets for particular problem.

Dataset	No. of hidden neurons used	Epochs	Overall Accuracy	Geometric Mean Accuracy	Average Accuracy
Ae	6	1000s	98.394	97.842	97.912
ION	40	2500	87.609	87.932	88.087
Iris	8	8000	94.572	93.021	93.367
Liver	11	3000	66.346	64.411	64.972
PIMA	10	800	73.781	73.673	73.689
VC	41	750	82.293	76.765	81.519
Wine	4	700	96.102	96.444	96.510

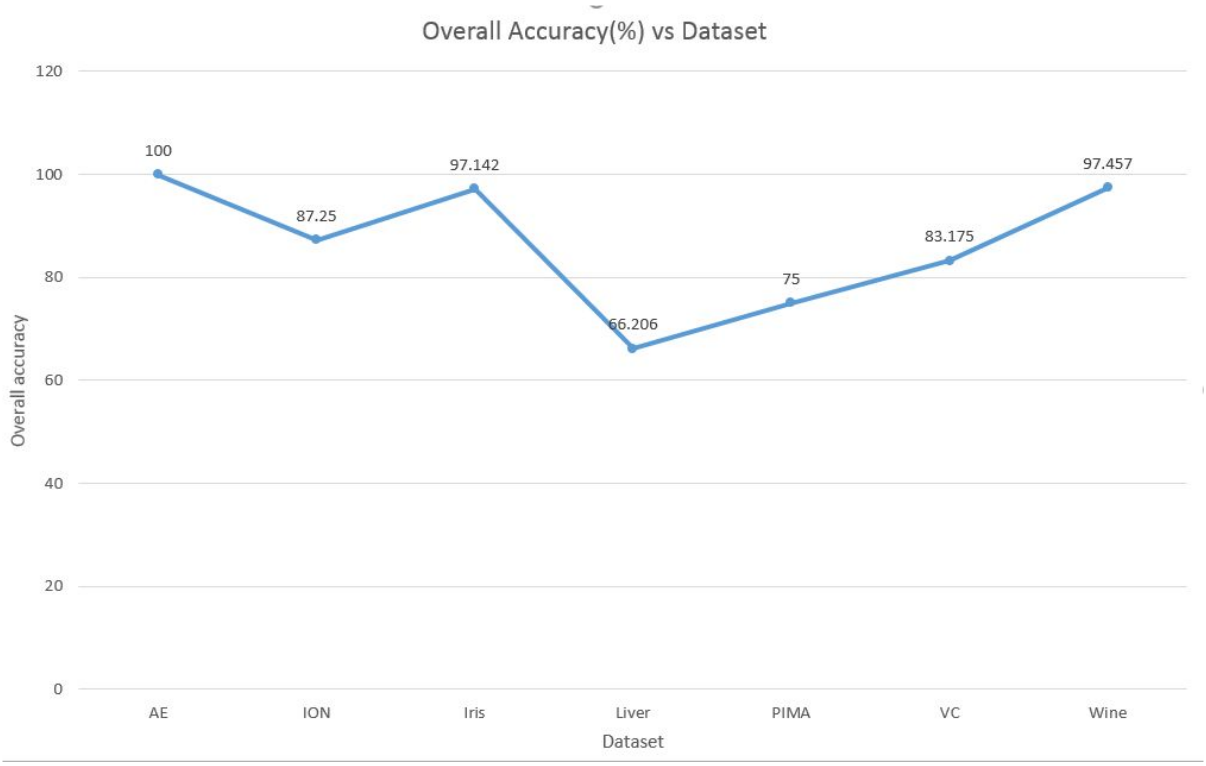
RBF Clustering LS

- Mit Naria

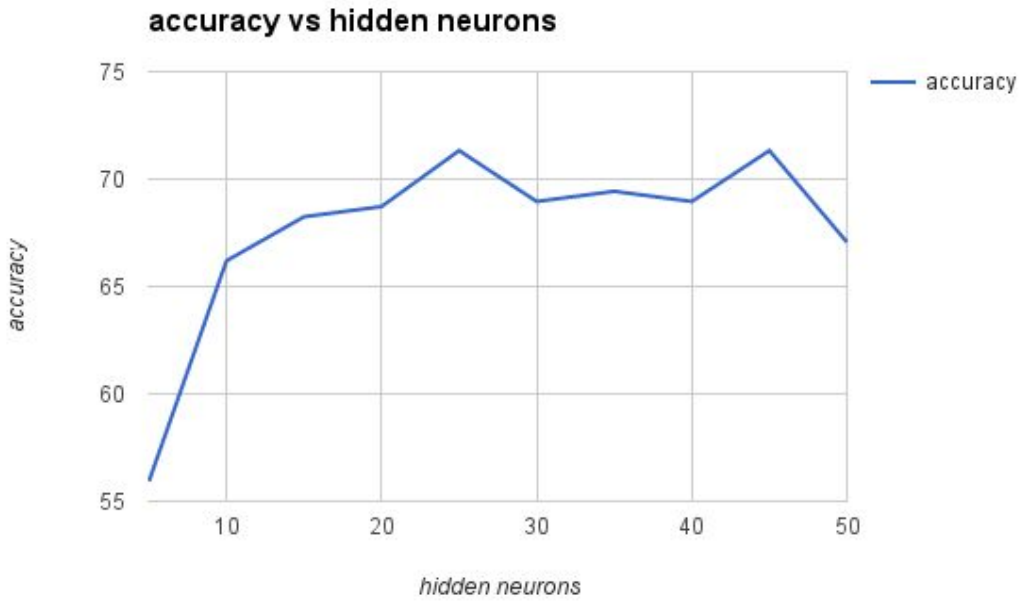
In this method , we use same technique as used in RBF for classification. We first start with few neurons which are randomly initialised from training. We increase hidden neurons and/or change spreads and centers of these units as learning process continues. Other method to initialise centers of these hidden neurons is to use K-means clustering to find K centers of these RBF hidden neurons.

After initialisation, we try to update weights , spreads and centers of these neurons in order to accurately cluster the data points. There is cost function which enables us to optimize these things. Usually cost functions require loss functions. I have used LS as loss function in RBF clustering. Also, in clustering , we use euclidean distance or Mahalanobis distance or Minkowski distance as measure to check whether a point belongs to cluster or not.

Dataset	No. of hidden Neurons	Overall Accuracy	Geometric Mean Accuracy	Average Accuracy
AE	6	100.000	100.000	100.000
ION	40	87.250	87.120	87.122
Iris	8	97.142	97.114	97.142
Liver	11	66.206	64.411	64.972
PIMA	10	75.000	73.210	73.336
VC	41	83.175	81.180	82.828
Wine	4	97.457	96.718	96.764



Accuracy vs Hidden Neurons graphs for dataset VC with training and testing with set1 data.



As you can see , initially when we add hidden neurons as accuracy increases as more neurons are able to cluster data correctly. As VC dataset contains 18 features and 4 possible classes , as thumb rule (No Of Inputs + No Of Outputs) * ⅔ = 14.66 As around 15 neurons , we are able to get good results. But, we get best overall accuracy when number of hidden neurons are 25 and 42.