

ASSIGNMENT -1

Neural Network for McCormick Function - Matlab

Course Code: CH6870 (Machine Learning for Process Systems)

Roll Number: EW21MTECH14003

Submitted by: Agathiyan Susil R

Question 1:

McCormick function (as shown below) is a combination of trigonometric and polynomial functions.

$$f(\mathbf{x}) = \sin(x_1 + x_2) + (x_1 - x_2)^2 - 1.5x_1 + 2.5x_2 + 1$$

$x_1 \in [-1.5, 4]$, $x_2 \in [-3, 4]$, Global minimum occur at $(-0.54719, -1.54719)$.

A NN is built based on this function, with the training data been taken randomly within the domain and “excel” was used to create the dataset (description as in below figure) initially.

```
% 1200 data points (x1, x2) have been picked up randomly with precision upto 3 digits.  
% These data points lie within the domain (x1 ∈ [-1.5, 4], x2 ∈ [-3, 4]).  
% The target output which is generated is normalized between 0 to 1 using the log-sigmoidal function.  
% Data points are imported from excel to matlab and stored here as ".mat" extension  
% 100 fresh unseen data samples in the same manner are created and imported for prediction.
```

Dataset Description

These 1200 data samples that are imported from the excel to the matlab, are further divided into training (70%), validating (15%) & Training (15%) sets. This allocation can be later modified, however by default matlab interface uses this percentage and divides the data randomly for each epoch.

Question 2:

After storing and importing the data as discussed in Question 1, using the “nnstart toolbox” we can train the model and generate the script for the same (how to use “nnstart” in detail will be discussed in Question 7).

```
x = input';
t = output';

trainFcn = 'trainlm'; % Levenberg-Marquardt backpropagation.
% Create a Fitting Network
hiddenLayerSize = [5 3] ;
net = fitnet(hiddenLayerSize,trainFcn);
% Setup Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
% Train the Network
[net,tr] = train(net,x,t);
my_NN_net = net;
save my_NN_net
% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)

% View the Network
view(net)
```

Script that is generated from “nnstart”

After generation of the script (as shown in Figure), the architecture of the model can be modified as per the requirement. Here since the number of layers used should be atleast 2, thus [5 3] and similar architectures are incorporated and analysed.

Question 3:

First, the model which is by default created with one hidden layer and 10 nodes is then modified and iterated with different architecture for the same dataset. After comparing all the results on the basis of performance and regression, the best suitable architecture is decided i.e., in this case it is [5 3] (see Figure a).

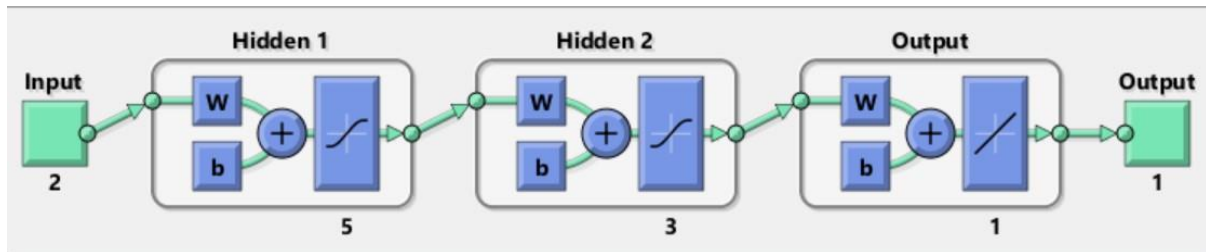
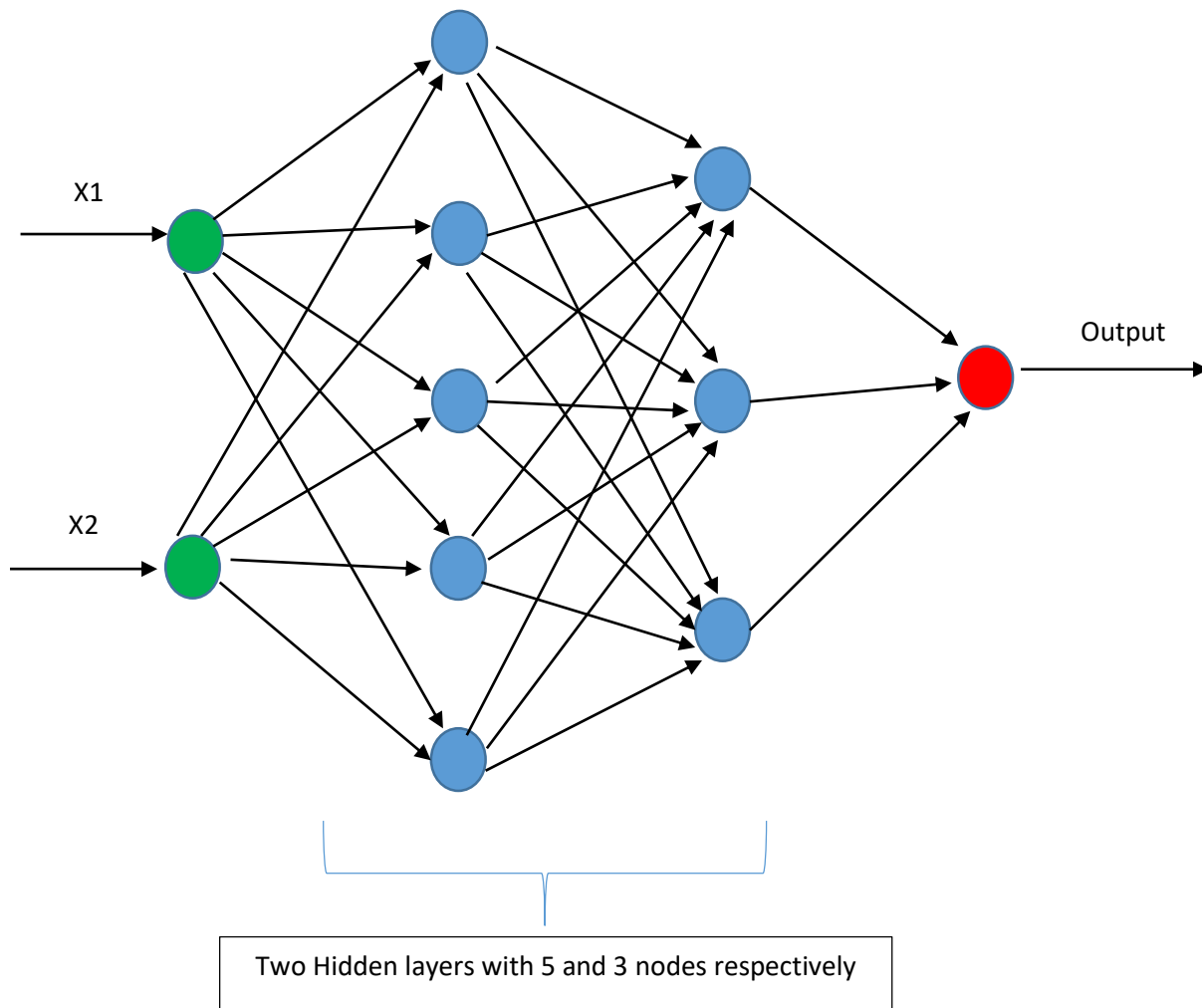


Figure a: [5 3] architecture

Here it is to be noted that, the algorithm (LM – Levenberg Marquardt) and the dataset are maintained the same throughout the architecture iteration. Different NN architectures considered are [10 5], [10 8], [5 3 2], [5 4], [10 5 3], [5 2], [10 3]. The chosen architecture looks as shown, with its respective value of weights as displayed in Question 5 (part e).



Question 4:

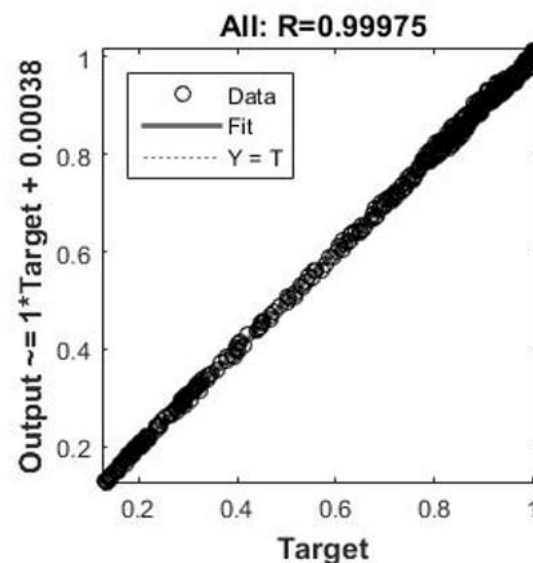
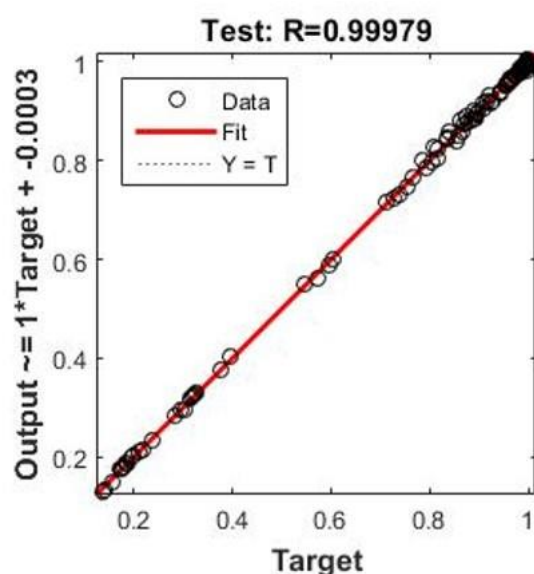
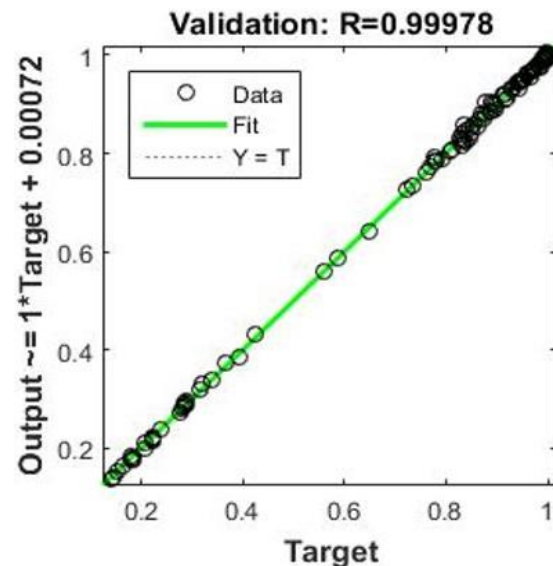
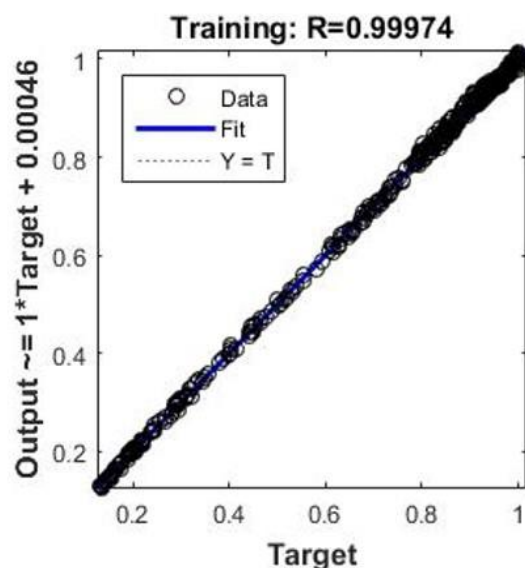
- a) **Network Performance:** This is the mean square error between the target and the output data. This value will be calculated by the model and displayed in command window as seen below.

```
Command Window
>> load('McCormick1.mat')
>> nnstart
>> McCormick

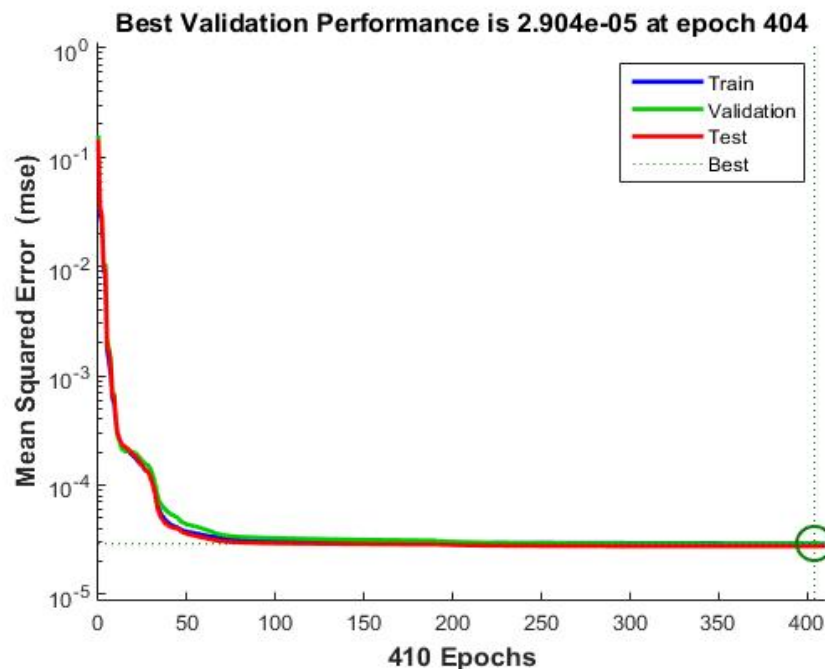
performance =

    7.1524e-09
```

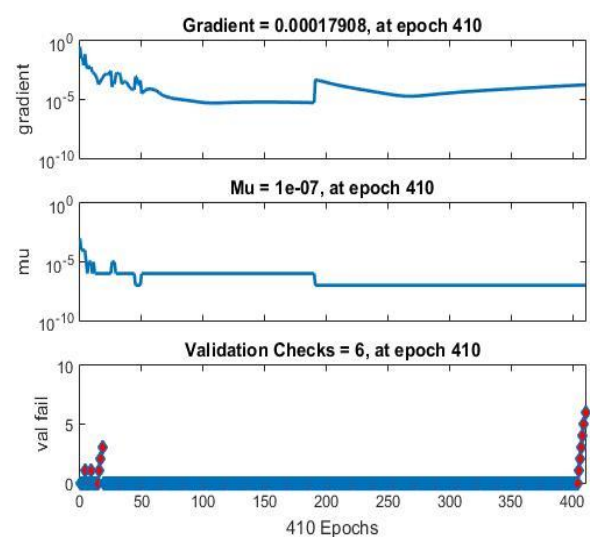
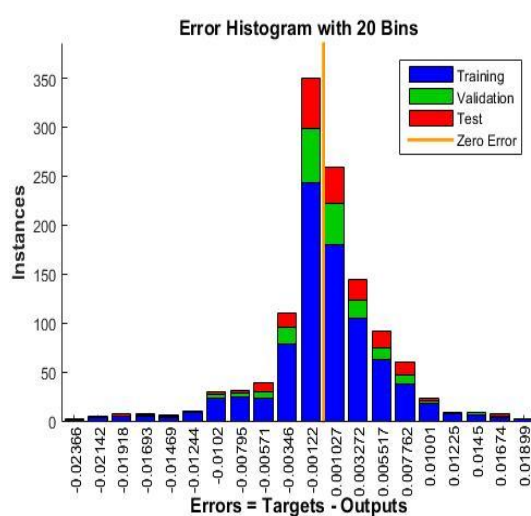
- b) **Regression plots:** Individual graphs generated for training, validation & test data sets simultaneously. "R" value shown against each plot shows how efficient the algorithm is.



- c) **Performance Plot:** Here, by generating this plot one could know for many epochs the model has run and at what value of performance it got terminated. In the below figure, validation performance is shown at the top.



Here we can see that the model was trained with 404 epochs, and it got terminated when it achieves the best validation performance i.e., a global minimum point in the validation curve. There are other plots (Error histogram & Training state) as shown below are generated alongside regression and performance plot for each specific run.



Question 5:

Now train the same network using **Gradient Descent** (GD) algorithm and draw comparisons between LM and GD in terms of rate of convergence (report relevant metrics to justify your observations).

Gradient Descent algorithm can be incorporated into the script by replacing “trainlm” with “traingd” in the script that is generated. Before doing that, the same script was iterated for atleast 10 times with LM algorithm and its important parameters are recorded as shown in Table 1.

Later, same of iterations is repeated with GM algorithm keeping all other hyper parameters as it was in the last case. Again these results and output parameters are noted as seen in Table 1.

S.No	Algorithm	Epochs	Performance	Predicted-Performance	Difference in Performance	% Difference	Regression
1	LM	500	8.93E-06	6.60E-06	2.33E-06	26.050	-
2		957	4.46E-07	5.56E-07	-1.10E-07	-24.599	1
3		148	7.78E-06	9.80E-06	-2.02E-06	-25.926	0.99994
4		238	2.98E-07	2.93E-07	4.93E-09	1.656	0.99999
5		410	2.90E-05	3.26E-05	-3.64E-06	-12.549	0.99979
6		64	4.08E-06	4.59E-06	-5.08E-07	-12.459	0.99996
7		113	2.71E-05	3.04E-05	-3.32E-06	-12.223	0.99966
8		1000	1.86E-07	2.03E-07	-1.74E-08	-9.340	1
9		137	3.82E-06	4.15E-06	-3.33E-07	-8.719	0.99996
10		94	6.04E-05	7.52E-05	-1.49E-05	-24.609	0.99936
Average			1.42E-05	1.64E-05	-2.25E-06	-10.272	0.999851
11	GD	1000	0.0422	0.0348	0.50987	17.536	0.50987
12		1000	0.0325	0.0246	0.69958	24.308	0.69958
13		1000	0.0288	0.0272	0.71105	5.556	0.71105
14		1000	0.0318	0.0246	0.68037	22.642	0.68037
15		1000	0.035	0.031	0.67242	11.429	0.67242
16		1000	0.058	0.0483	0.10984	16.724	0.10984
17		1000	0.0316	0.0258	0.64762	18.354	0.64762
18		1000	0.0562	0.04664	0.2347	17.011	0.2347
19		1000	0.0222	0.0201	0.78726	9.459	0.78726
20		1000	0.0204	0.0144	0.83966	29.412	0.83966
Average			0.03587	0.029744	6.13E-03	17.243	0.589237

Table 1: Result Comparison Between LM & GD

Note: The architecture used here is [5 3], which gave promising results with LM algorithm. And the input data contains 1200 samples which was used to train both the models. Then a separate matlab script was written for predicting the performance of 100 unseen data in the both the cases and recoded the “predict_performance” as displayed in Table 1.

By doing so, the data and plot which are generated in each case can be used to compare among the algorithm. Some of the comparisons among LM and GD are discussed below:

- a) Firstly, a function or system gets validated only if it reaches or achieves convergence. Similarly, here **one way of interpreting convergence is from number of epochs** the algorithm has run and stopped to give its result. As we could see from Table 1 that, **GD algorithm doesn't get converged** (number of epochs run is always 1000) at any of the iterations while **LM achieves the convergence** most of the times.

Conclusion: So when the same algorithms are said to operate on very large dataset, then GD takes long time to converge or maybe it doesn't converge at all whereas LM algorithm the results come together and provides the desired.

- b) Converging of results are directly linked to the performance (MSE between target and actual output) of the algorithm. In light of that, we can see from Table 1 that the **average performance value of LM algorithm is very less when compared to GD algorithm** (almost 2500 times). It means that the deviation from the target is very less in LM when compared to GD.

Conclusion: LM works closely, captures and fits almost all the samples efficiently than GD.

- c) The above argument can also be supported by looking into Fig A & Fig B. Here in Figure A we could witness that the target plot and actual output plot of predict data almost overlap one over the other, sighting that the error or MSE is very less and hence the performance.

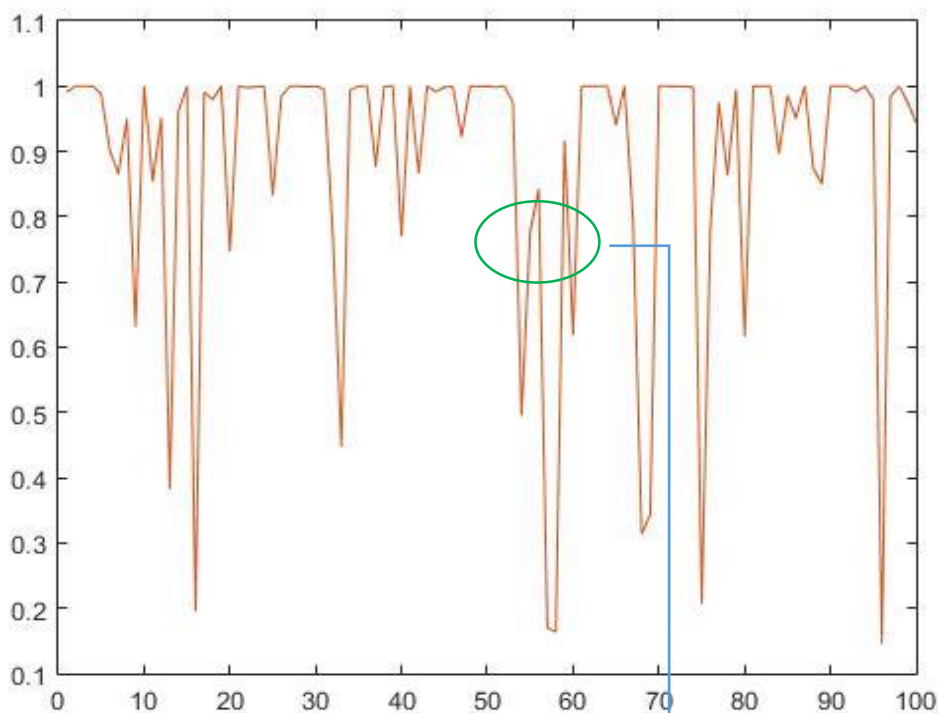
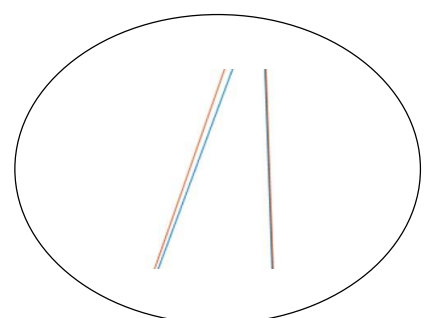


Figure A: Target vs Output of predict data using LM



While in Figure B, it clearly shows that there is a significant difference between the target and the actual output of predict data. The values of differences in performance (during training & Predicting) with their percentage difference are captured and recorded in Table 1, where the **average % difference in performance of GD is almost 1.7 times that of LM.**

Conclusion: Eventhough when the model fits the samples during training, validating & testing- it should perform similarly during prediction. The inference from the data in Table 1 and the inference from the figure A & B resembles the same indicating the advantage of LM over GD algorithm.

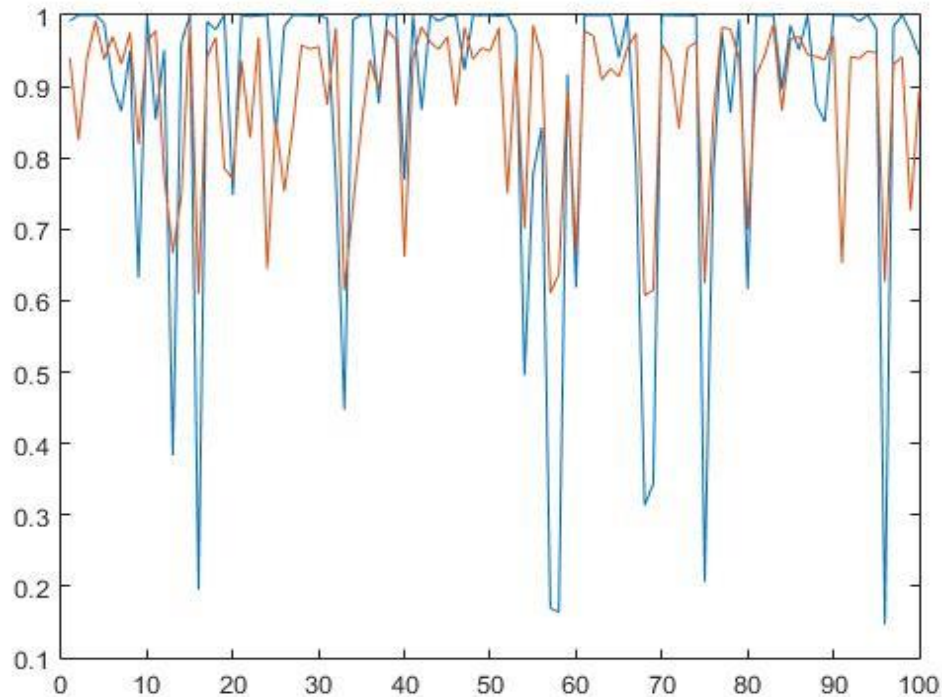


Figure B: Target vs output of predict data using GD

- d) Regression (R) is the other crucial parameter, with which the algorithms can be compared before getting into prediction phase. By looking into the Figure C, it is understood that **LM algorithm with $R = 0.99$ (for training, validating and testing samples) is very robust towards fitting.**

Whereas, in the Figure D which is obtained by **incorporating GD algorithm- regression value ($R = 0.71$) is relatively lower** when compared to earlier case.

Conclusion: Take away from here is that not all samples get fitted by using GD algorithm, besides almost all the data samples are captured using the LM algorithm which hence gives better prediction results as seen earlier.

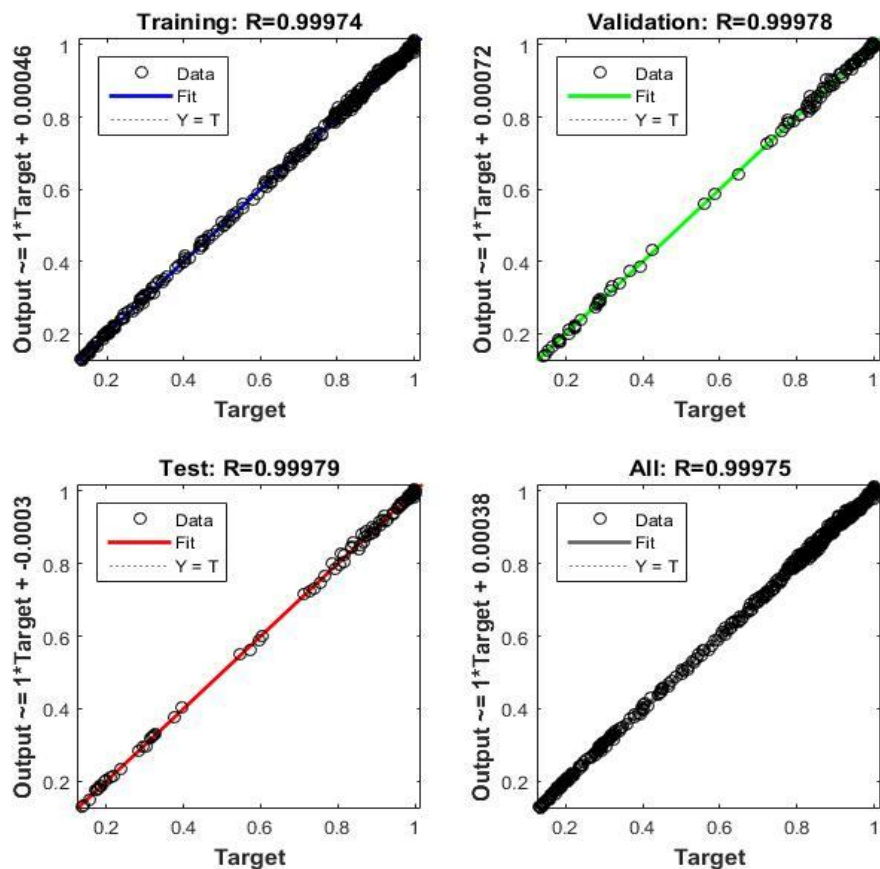


Figure C: Regression plots from LM algorithm

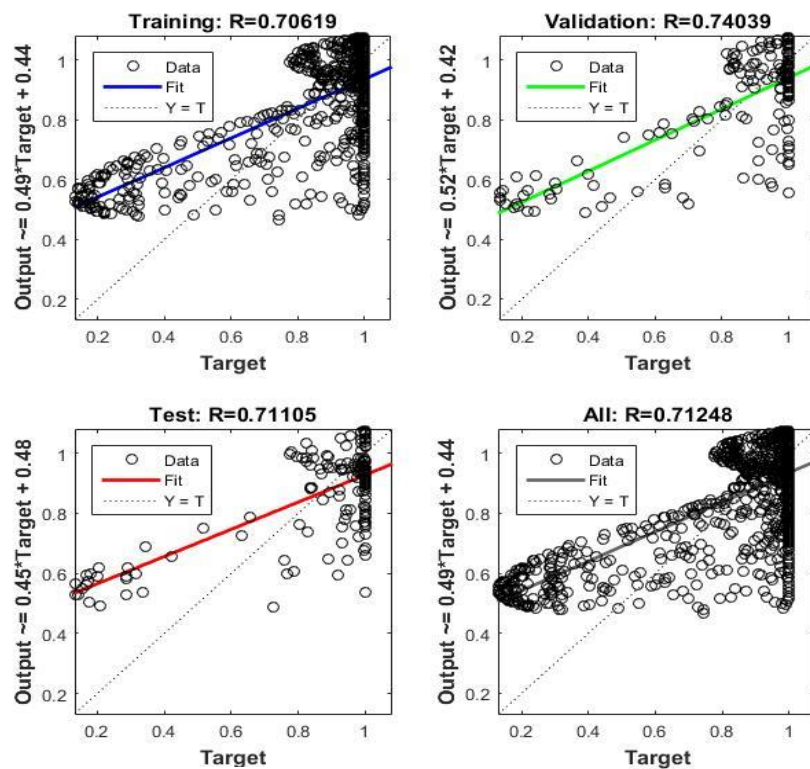


Figure D: Regression plots for GD algorithm

- e) Apart from these comparisons, when we seek into the NN of both the algorithms it can be seen that there is significant difference in the respective weights and biases that are learnt and improved in order to get the desired output with least MSE.

All these variations in performance, regressions are because of these weights and biases that differ by large margin respectively as shown below.

W1	-1.10792	-1.40547
	2.850052	-3.51806
	-2.26039	2.900677
	2.796655	-3.60337
	-0.71422	-0.9094

W2	-7.97261	-0.63062	0.620058	1.037211	11.22308
	-7.43085	1.754266	1.399804	-0.91036	10.60427
	-7.21857	2.024488	1.401398	-2.51555	10.41176

W3	-1.13543	1.803935	-0.79949
----	----------	----------	----------

These are the final weight matrix of LM algorithm (of an iteration) with [5 3] architecture.

W1	3.016096	0.85125
	-0.42194	3.095471
	-2.92431	1.100409
	-2.05165	-2.29208
	2.079244	2.362667

W2	-1.24912	1.090954	0.400999	-0.11919	0.334003
	-0.43961	0.000468	1.061064	0.51652	1.087857
	0.394576	-0.09414	-0.36813	0.728303	-1.52884

W3	-0.43003	-0.44569	-0.72447
----	----------	----------	----------

Final weight matrix of GD algorithm (of an iteration) with [5 3] architecture.

Conclusion: From the above we can see that, **there is significant difference in the weights especially of W1 & W2.** The same trend is observed in biases too. These all together consolidate and show us the variation in results for different algorithms and hence allows the user to choose the best suitable one.

Question 6:

For Prediction 100 fresh labelled data ("Predict.mat") are taken and given as input to the model. By doing so (see Figure (i)), output along with the prediction-performance are obtained.

```
close all

load my_NN_net
net= my_NN_net;
view(net);

load('Predict.mat','F_input','F_output');
n=100;
f_in= F_input(1:n,:);
target= F_output(1:n,:);
f_out= net(f_in);
errors= gsubtract(f_out,target);
w1=net.IW{1,1};
w2=net.LW{2,1};
w3=net.LW{3,2};
b1= net.b{1};
b2=net.b{2};
b3=net.b{3};
x= linspace(1,n);
plot(x,errors)
plot(x,target,x,f_out)
prediction_performance = perform(net,target,f_out);
```

Figure (i): Prediction Script

These outputs which are obtained are then compared with the actual target (see Fig A – Question 5). By comparing it is very much evident that “LM” algorithm with [5 3] architecture is best suitable for the task that was undertaken (McCormick Function). From this script respective weights and biases which are the major building blocks of each layer can also be tracked.

Question 7:

1. How to train, validate and test an ANN for a given dataset using nnstart?

Introduction:

“nnstart” is a command in matlab which enables one to access the inbuilt neural network features like Neural Net Fitting app, Neural Net Pattern Recognition, Neural Net Clustering app & Neural Net Time Series app. These branches are used for different applications, and here one such of kind is considered where “Fitting app” is utilized for building a NN for specific cause. This part of the report elucidates how to enable and use “nnstart” with an example.

Procedure:

Stepwise approach is discussed briefly, following the same will end up in building your own such kind of NN in matlab.

Step 1: For anything to start, data is required. So similarly there should be some labelled data which should be loaded in matlab (see Fig 1) to train our NN for a specific task. Matlab gives the user many provisions for loading the data, one such is directly importing from the excel.

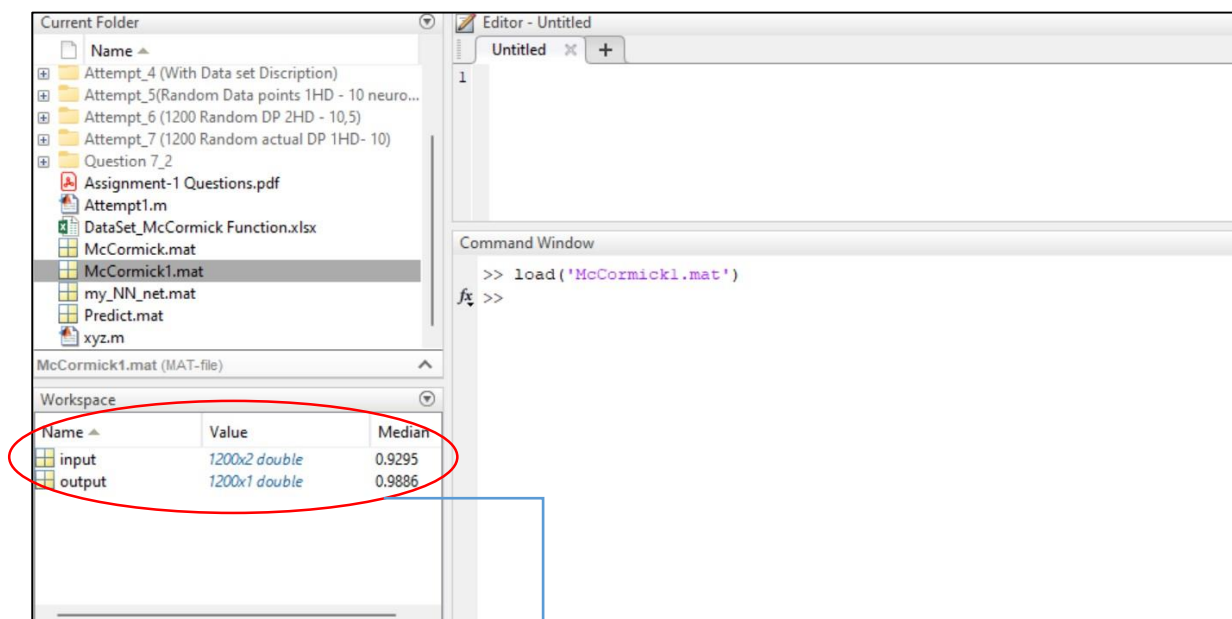


Figure 1: Dataset load

Here as seen, “input & “Output” data are created and stored as “. mat” file. The matrix architecture is provided aside of each showing us the number of data been imported for training purpose.

It should be noted here that how the data is stored in the variable “input” and “output”, either row ways or column. This further helps us in the upcoming steps where these are assigned. Here the data with which I have handled has 1200 samples which are stored in row. The “output” which again have 1200 points are normalized using the log-sigmoidal function and stored in rows.

Step 2: As shown in Fig 2, type “nnstart” in the command window. This is basically a command which gives rise to a pop up window (see Fig 3) where the different features of NN can be witnessed with one being the “Fitting app”.

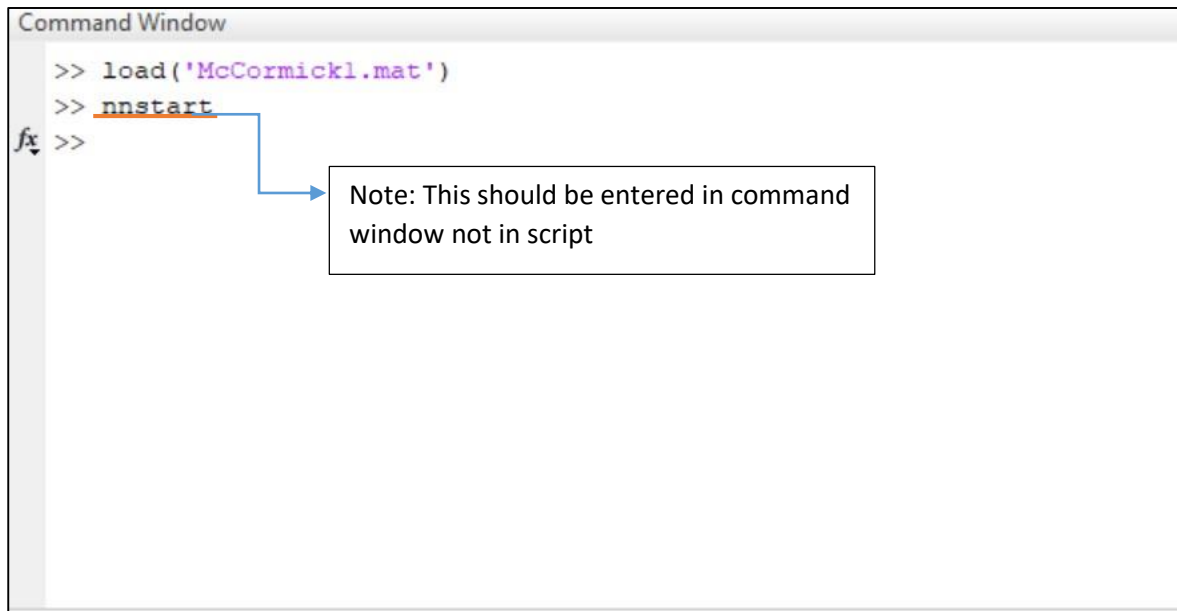


Figure 2: Initiating nnstart

In continuation of this, the pop up raises with different wizards and click or choose “fitting app” out of the options.

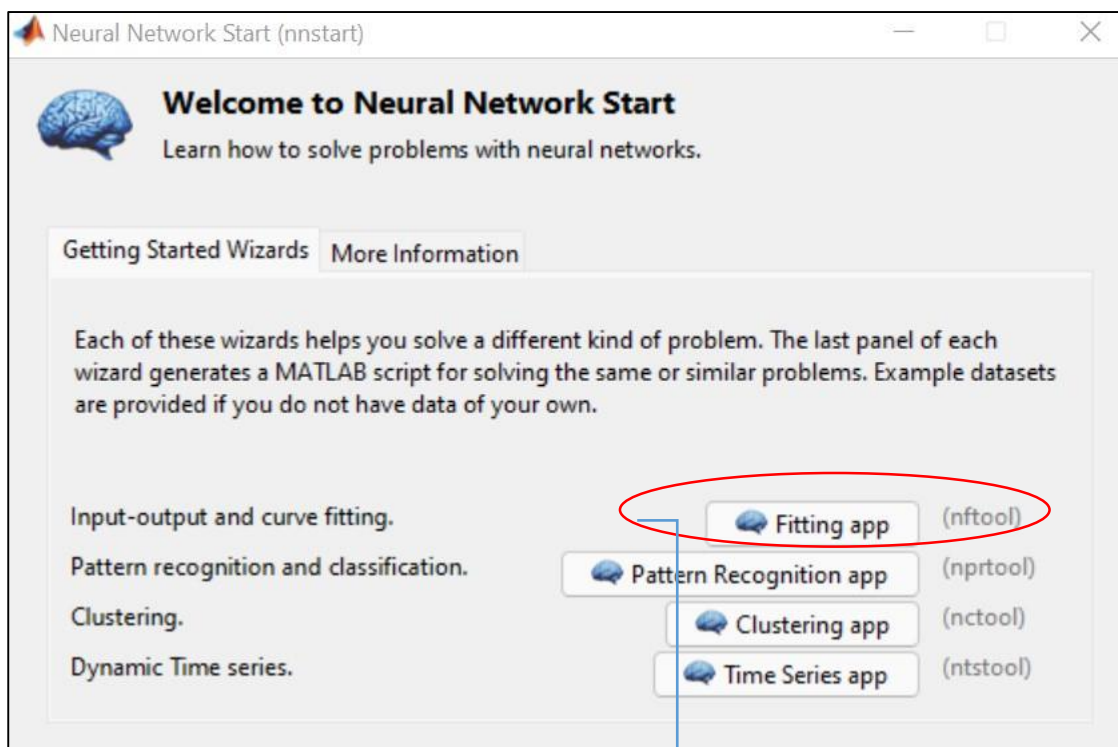


Figure 3: nnstart pop-up window

Step 3: After choosing “Fitting app” it takes us to the next window where a brief of introduction of “fitting app” and neural networks will be depicted as shown in Fig 4. Here just “click next” which will take us to the next step.

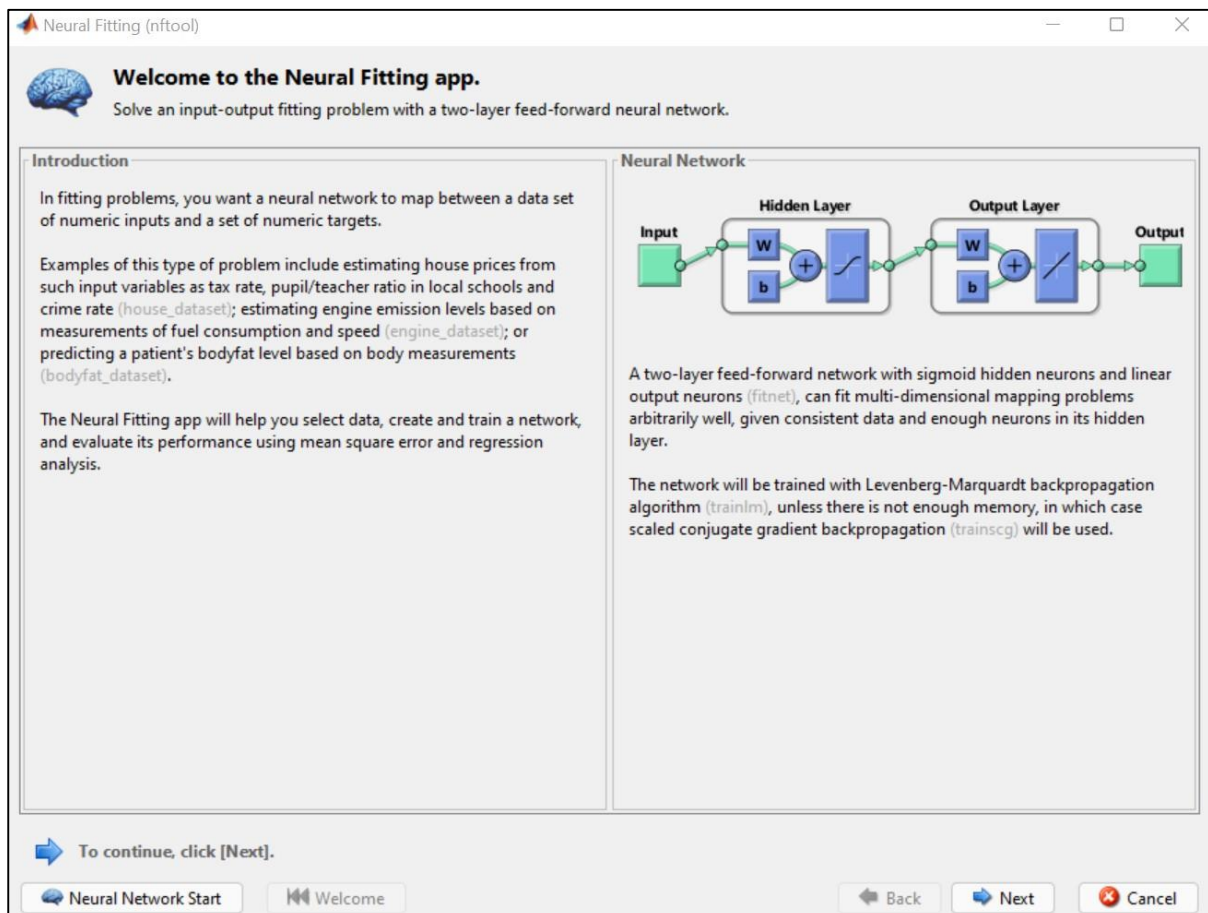


Figure 4: Introduction to Fitting app

Step 4: Now in this window as shown in Fig 5, using the data that was loaded (step 1) in workspace input and output should be assigned accordingly.

Note: The data that are assigned should be cross verified whether it contains the information in rows or columns, accordingly “matrix columns” or “matrix rows” should be chosen.

These data that are provided are customized. There is also a provision for loading the inbuilt data from the matlab as shown in Fig 6. It can be accessed by clicking on “load example data set” at the bottom left of the Fig 5.

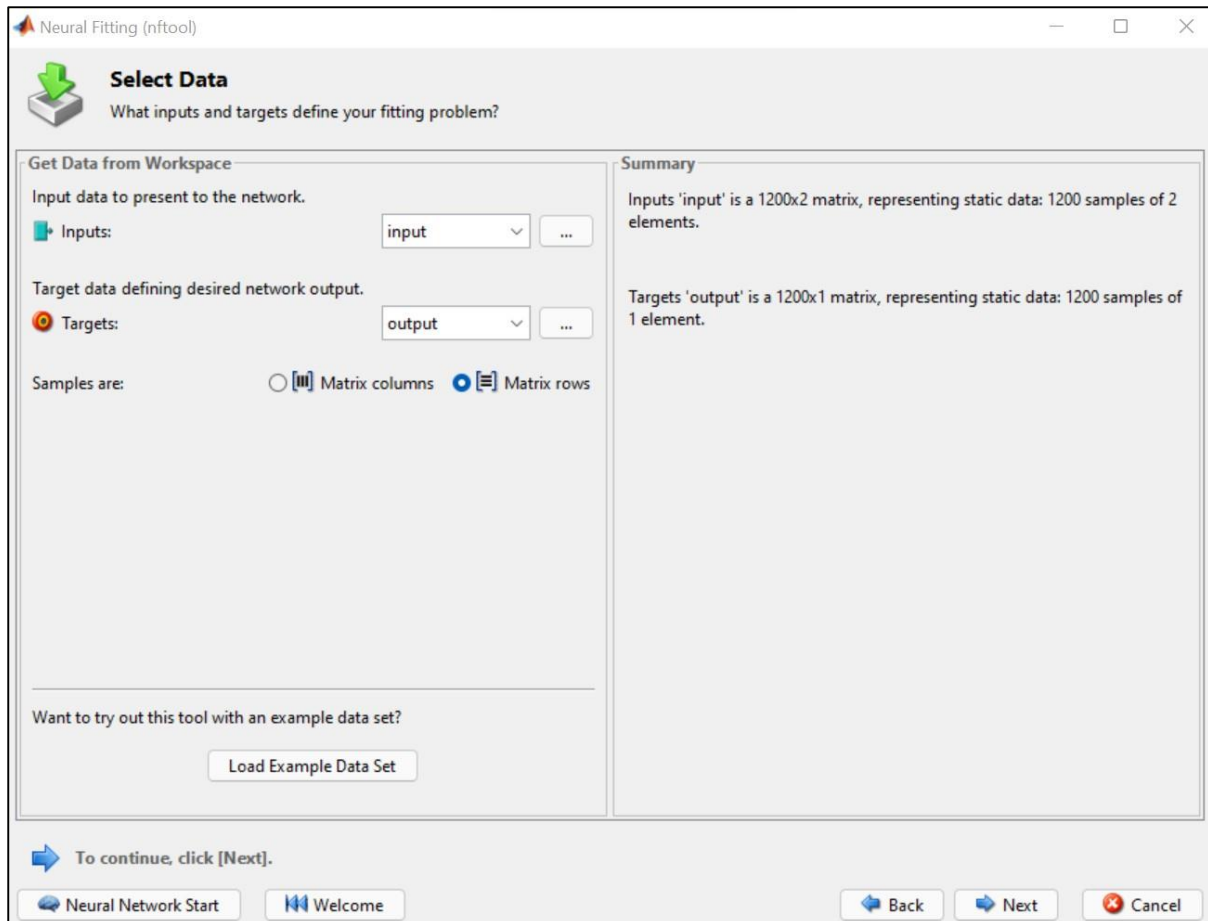


Figure 5: Selecting data

These are the inbuilt data sets provided by matlab. It can simply be incorporated into our network by choosing the desired and clicking on “import”.

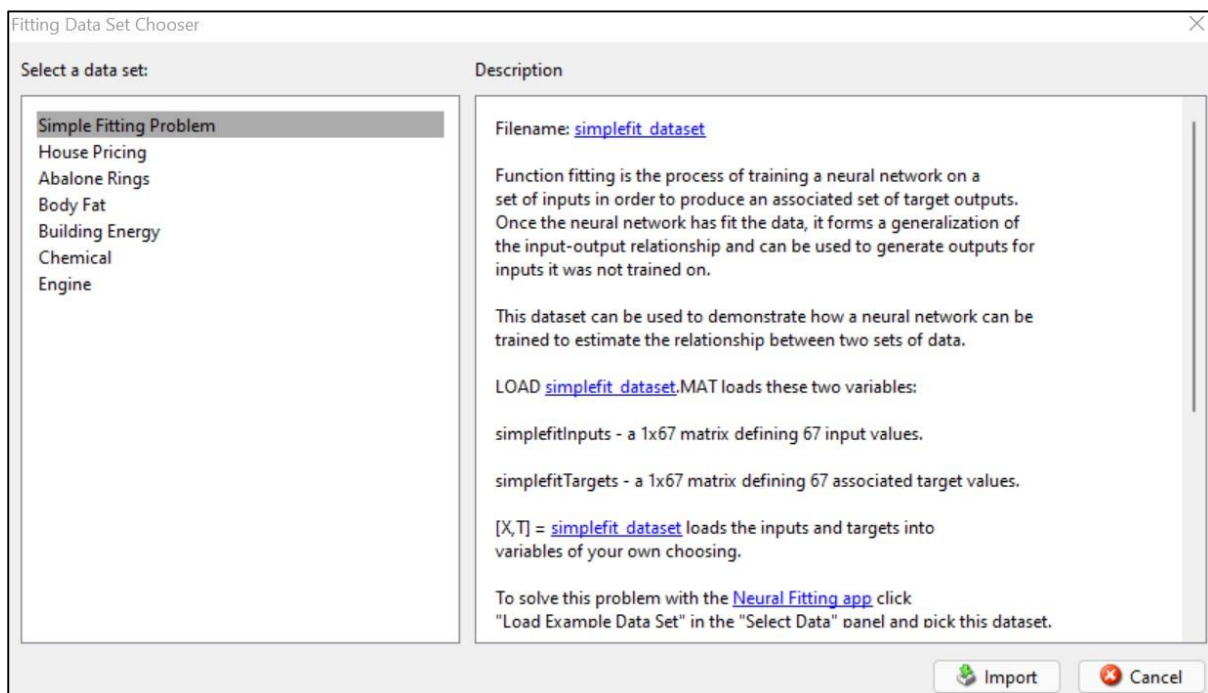


Figure 6: Loading Inbuilt data

Step 5: After loading the data, now the data should be divided into three parts- Training, Validation and Testing. Here in the Fig 7 we can see that by default the division percentage among the 3 is displayed. Division among the validation and testing data can be changed, but it is advised to go with this universal division percentage.

Note: As I have taken 1200 data samples, which are hence divided into 3 data sets now as per the percentage of division.

By clicking “next” it takes us to the next window, for further proceedings.

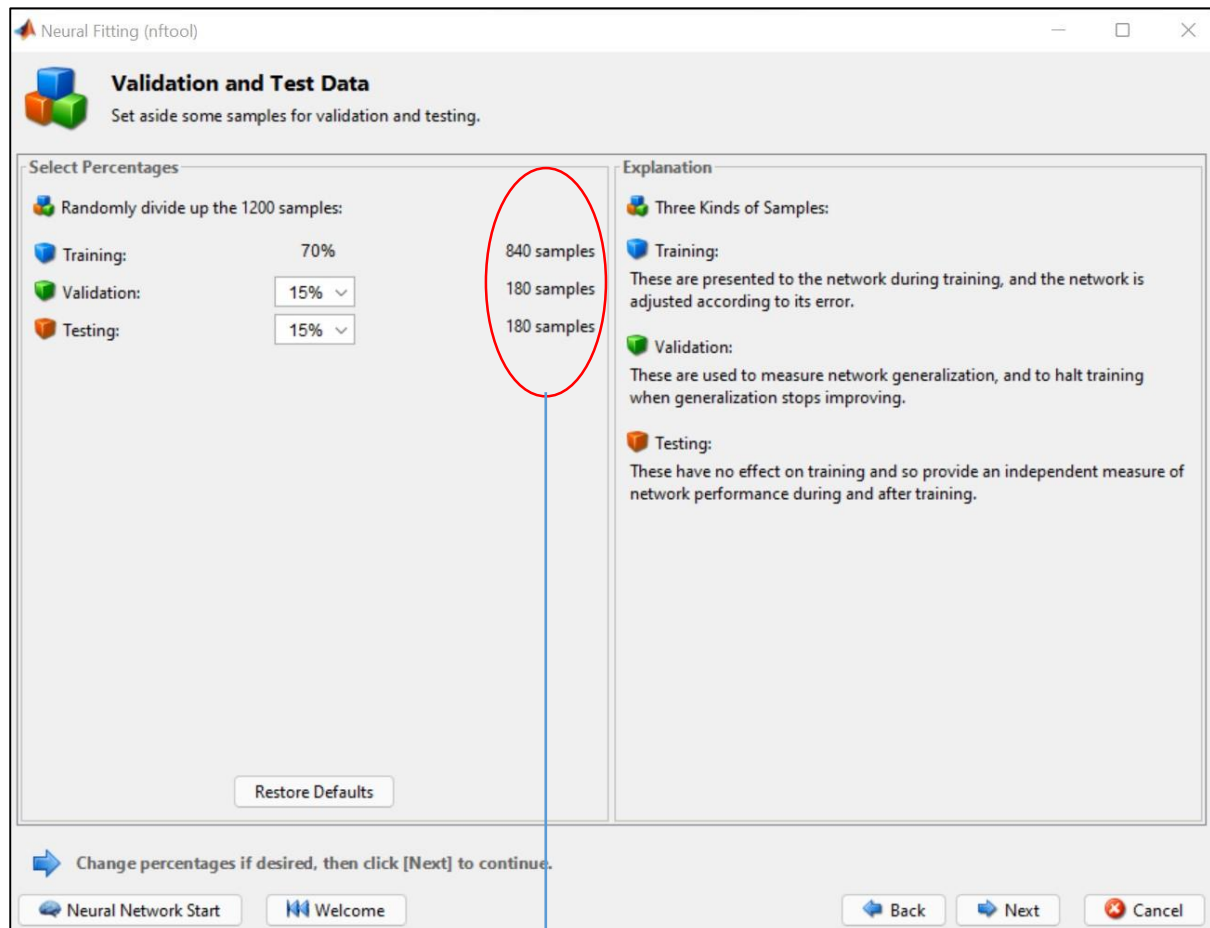


Figure 7: Data division

Here, the data gets randomly divided and for each epoch it will get shuffled among the 3 data sets however the division percentage will remain the same throughout a single run (i.e., train once) until it is changed here or in the script.

This random division of data can also be considered as a hyper-parameter, as it also plays a role in efficient training of the neural network.

Step 6: This step is one of the crucial one, where the architecture of the NN is decided for an initial run. As seen in Fig 8, there is provision to choose number of neurons in the hidden layer. However, this provides us to do so for only one hidden layer.

Number of hidden layers can be increased according to the need by modifying the script (see Fig 19 & Fig 20). By clicking “next” it moves to next window, or if changes required to be done in any of the previous window, then click “back”.

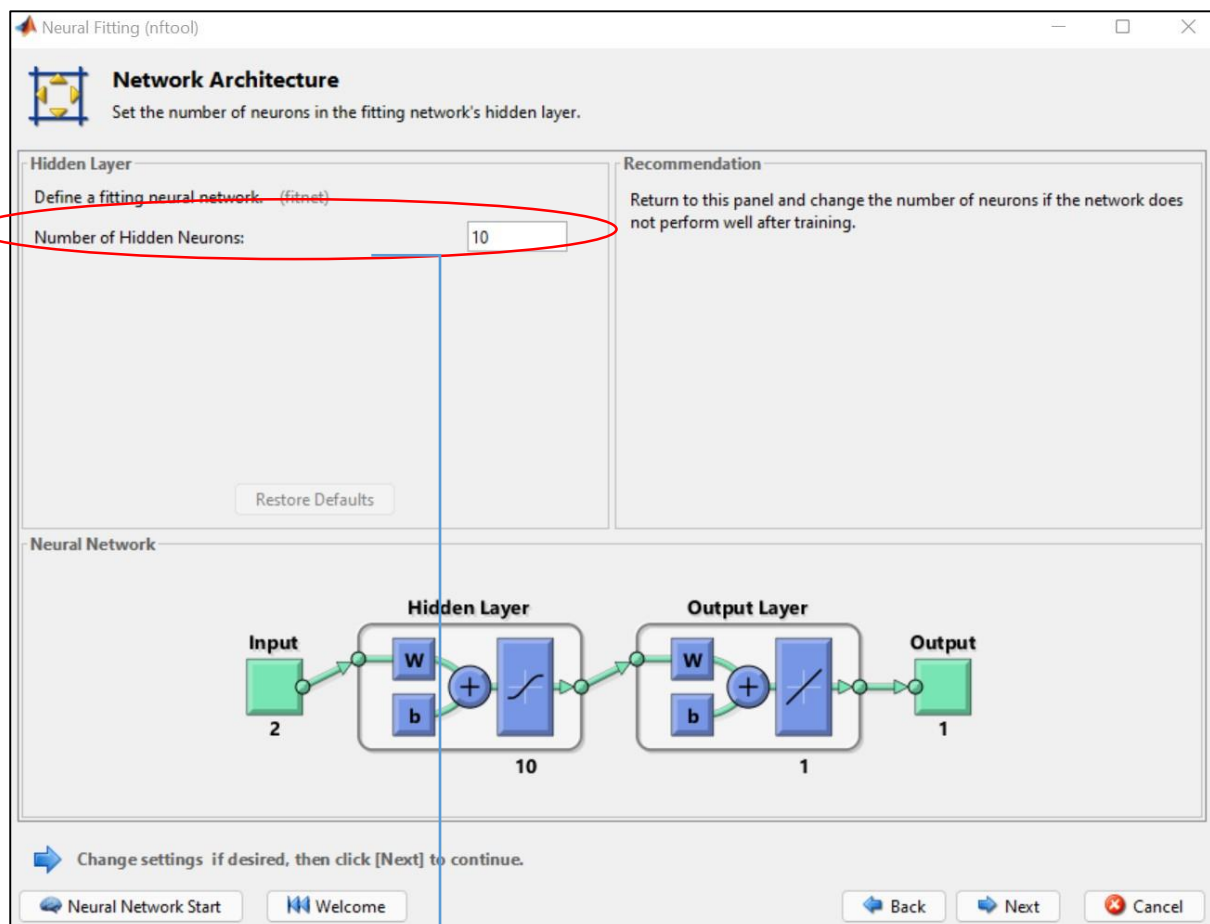


Figure 8: Defining Architecture of NN

Number of hidden neurons can be changed here, by default it displays 10. A demo neural network is shown at the bottom of the screen.

Step 7: Now the NN architecture is defined, so here in this step we train the model by clicking on “train” as seen in the Fig 9. Before doing so, user has to ensure the training algorithm that he has to incorporate for training.

There is drop down on the top of this window (Fig 9), where there are 3 algorithms to choose upon. By default, LM (Levenberg- Marquardt) is chosen by the interface. There are more than 3 algorithms which can be explored and incorporated (see Figure 22 & Figure 23).

“Next” icon is disabled here, as after deciding upon the algorithm the interface asks the use to run an initial train. After doing so, “next” icon will be enabled.

This window also gives us the definition of MSE (mean square error) and R (Regression), which would give us more understanding about the results we obtain after training.

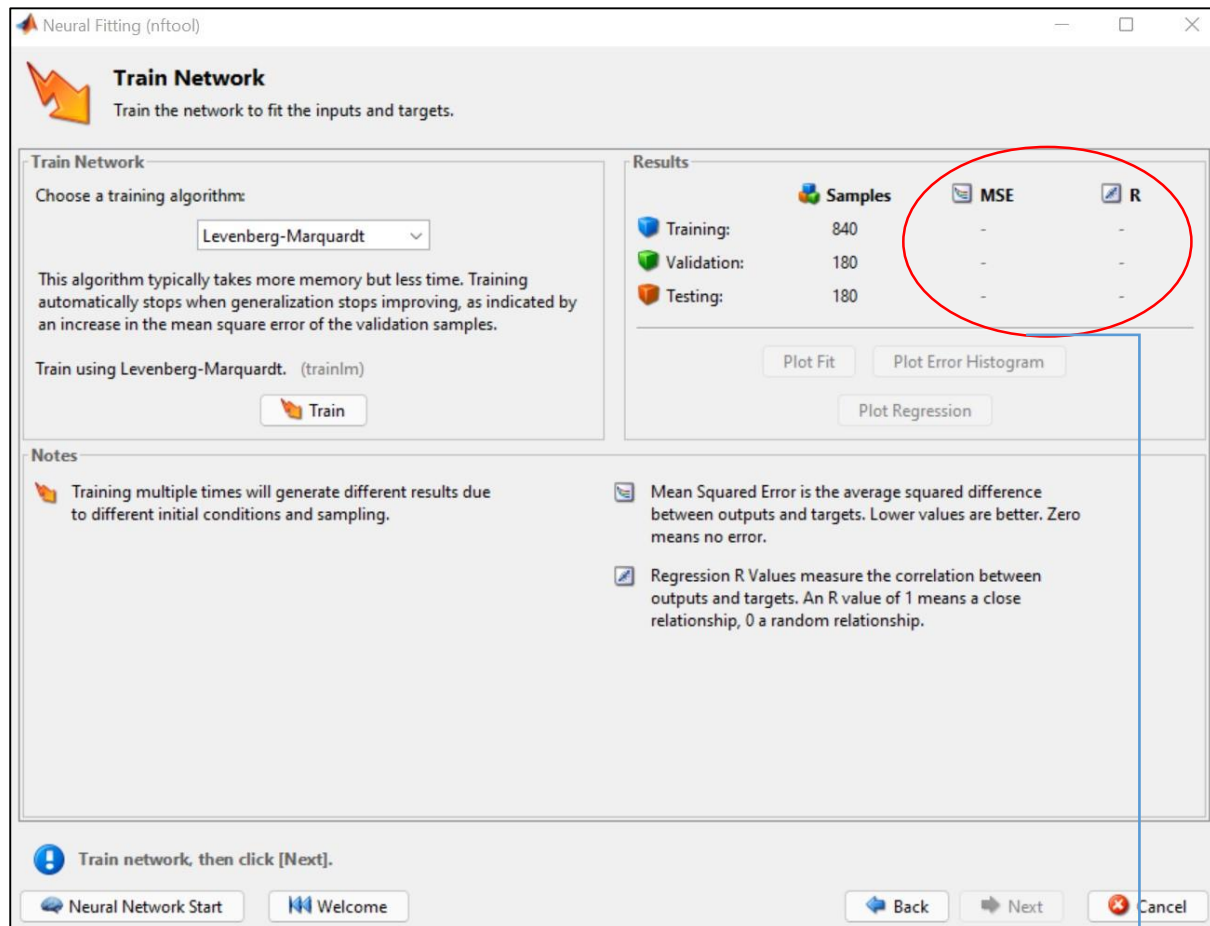


Figure 9: Training The Model

Note: Once the initial training is done, MSE and R values for each data set will be displayed here for interpretation

Step 8: After clicking on “train” in previous step, a pop up raises as shown in Fig 10. This gives the glance of the setup that was followed from step 2. It provides us results for the train network along with the access to different other output parameters including the plots.

Here, under progress and plots one can infer how the model has been trained in that attempt. By clicking on the respective plot (i.e., the “performance”, “training state”) it get displayed and can be used for interpretation.

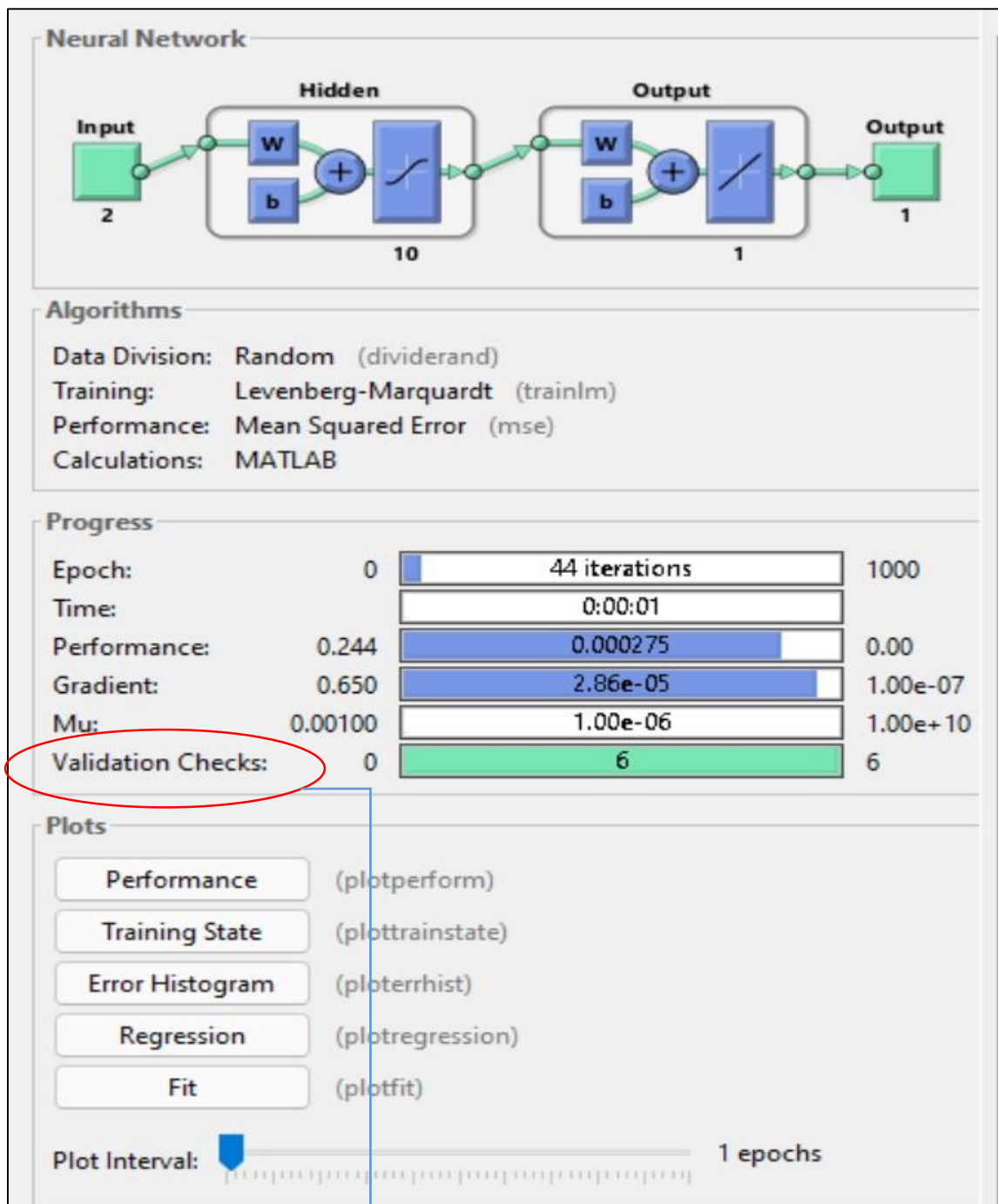


Figure 10: Result Glance Window

This "Validation check" factor says that the training ends if the validation failure occurs consecutively for 6 times. The same can also be seen in "training state" plot.

Step 9: After training and getting the result, “next” icon is enabled in figure which by clicking takes us to a new window as shown in Fig 11. Here there are multiple options to remodel the NN that was used earlier.

Train again: This option is used to retrain the same architecture and data set, which by itself gives different results as the shuffling of data re-initialization takes place.

Adjust Network size: By clicking on this it takes us to the step 6, where the number of neurons can be varied and again following the further discussed steps.

Import large data set: Enabling this, take the user to the step 4 indeed, and again the steps further should be followed.

If the user decides to not use these options, then by clicking “next” at the bottom of Fig 11 takes the user to next window.

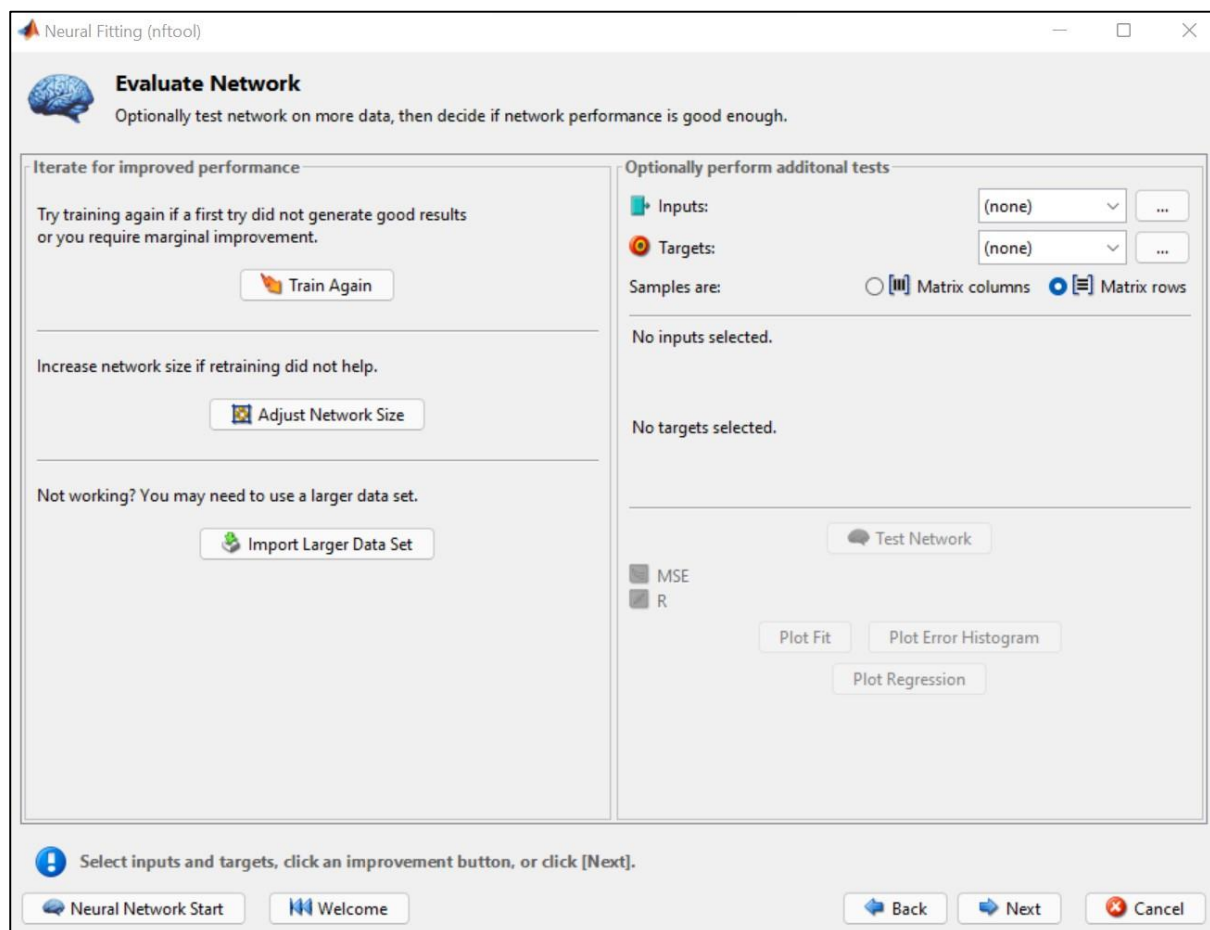


Figure 11: Network Evaluation

Step 10: The next window as shown in Fig 12, gives the Simulink diagram & graphical diagram options, which can be utilized based on the demand of the problem and the user.

By clicking “next” it moves on to new window and at the same if there is something needed to be modified then “back” tab can be used which will take the user to previous window.

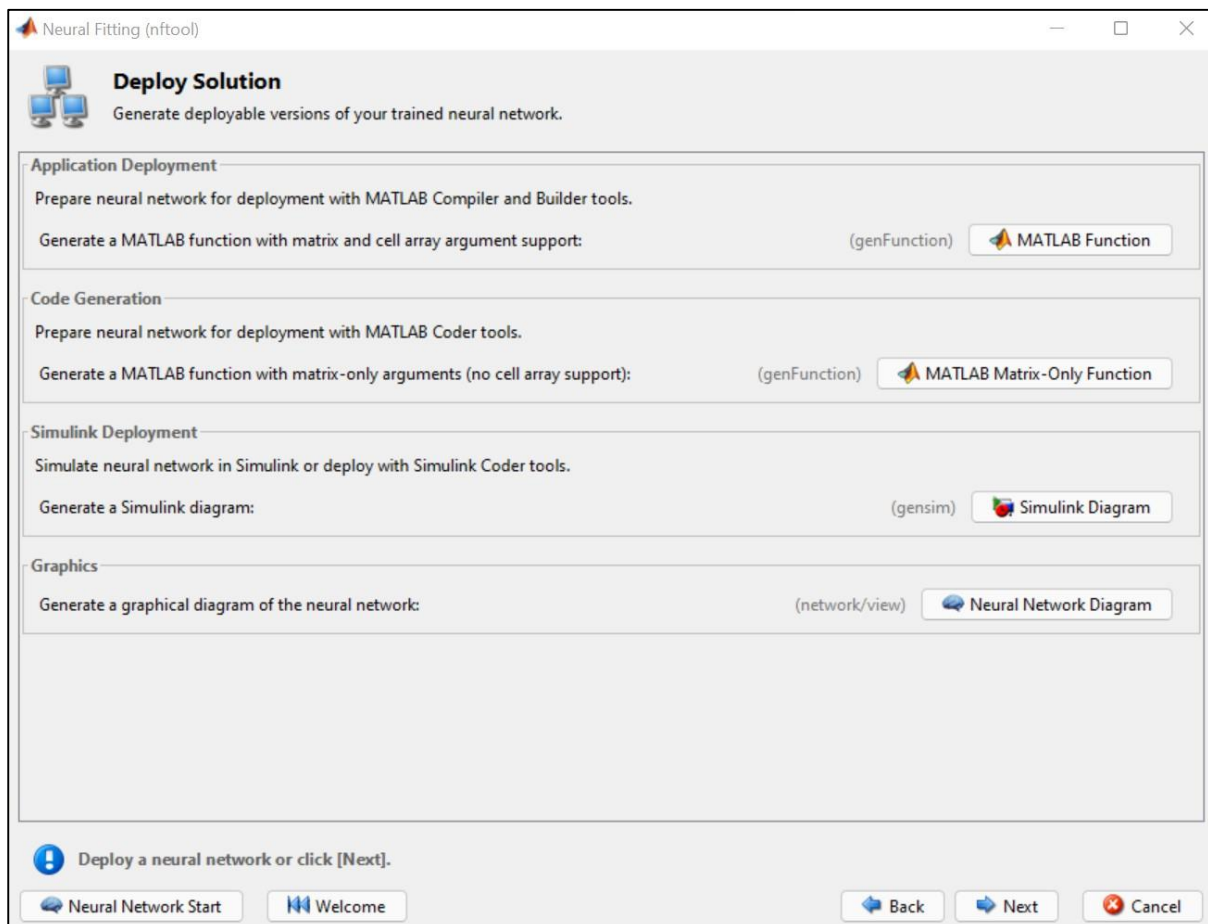


Figure 12: Simulink and Graphical diagram of NN

Step 11: At this step the user reaches the final window of “nftool”. This window (see Fig 13) provides the user to explore the “simple” as well as the “advanced” script of the NN that was been modelled and used all along from the beginning.

When the either of the script is chosen, it automatically displays the code with necessary comments in the script window of matlab.

One can see the difference between the simple and advanced script in the code itself, as the later includes some detailed information including the “error histograms” plot and other minuscule things.

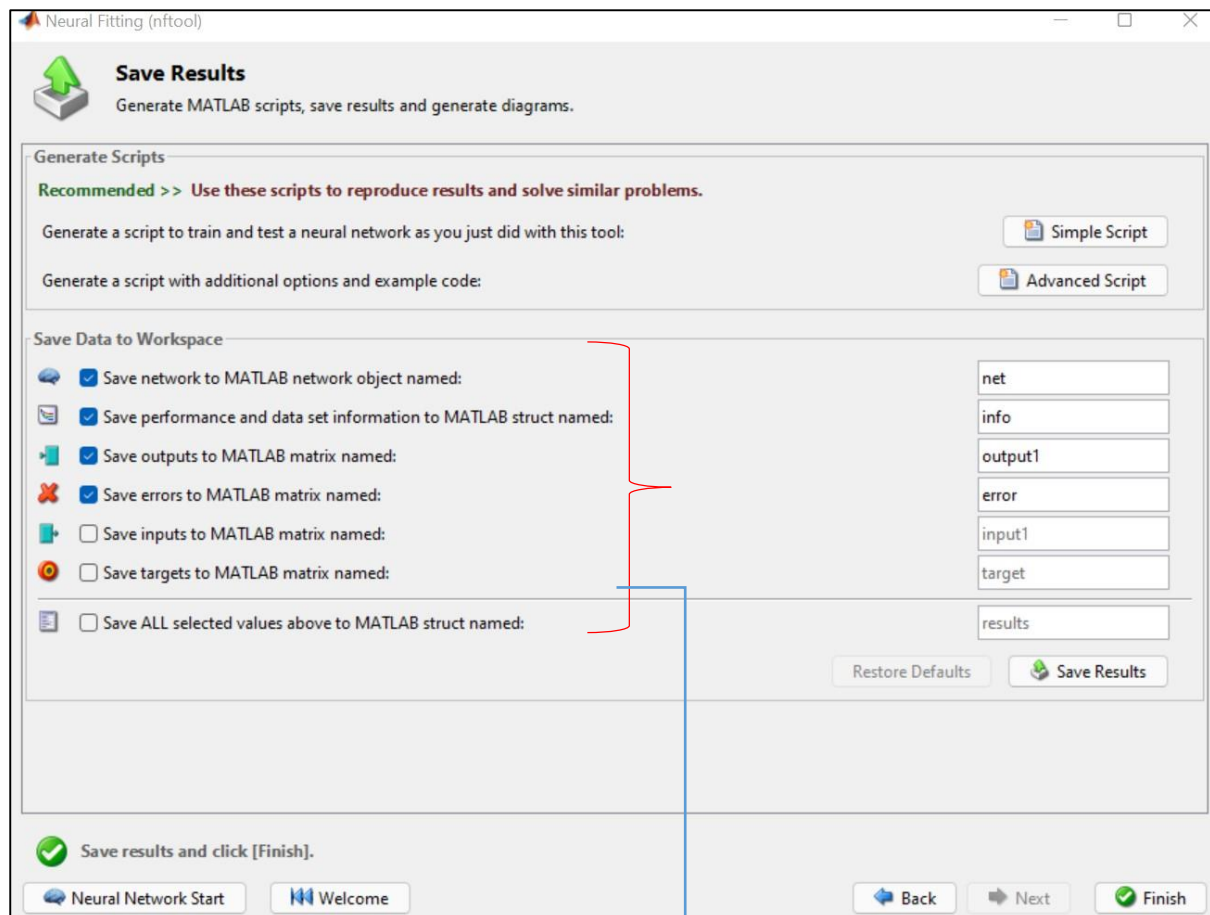


Figure 13: Script generate & Save Results

Saving results of necessary in workspace is controlled here in this window, where by enabling and disabling certain things gets reflected in the workspace.

Modifying the NN:

By following these steps (From Step 1 to Step 11) one can construct a simple NN (NN with one hidden layer) in matlab and extract its results (one such is shown in Fig 14) and interpret it accordingly. Similar to Figure 14, there can be many interpretations that can be drawn from other graphs ("Error Histograms", "Regression" & "Training State") as well.

Further steps from now will discuss how to modify the NN (i.e., increasing the number of hidden layers, switching to different algorithms) using the script that is generated in step 11.

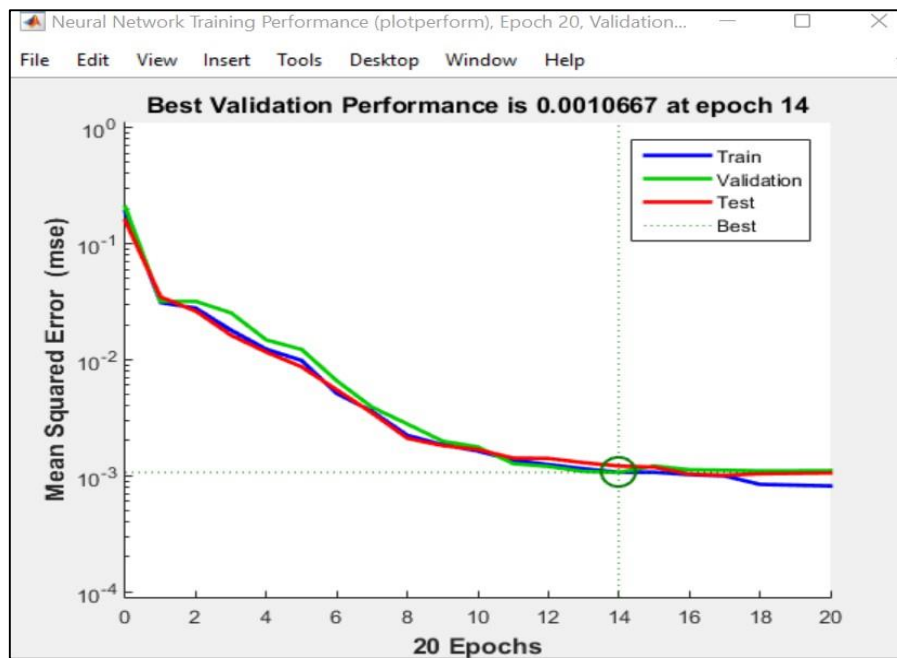


Figure 14: Training Performance Plot

Step 12: Before modifying and running the script, it should be saved at a particular domain. “Save option can be found at the left corner of the window as shown in Fig 15.

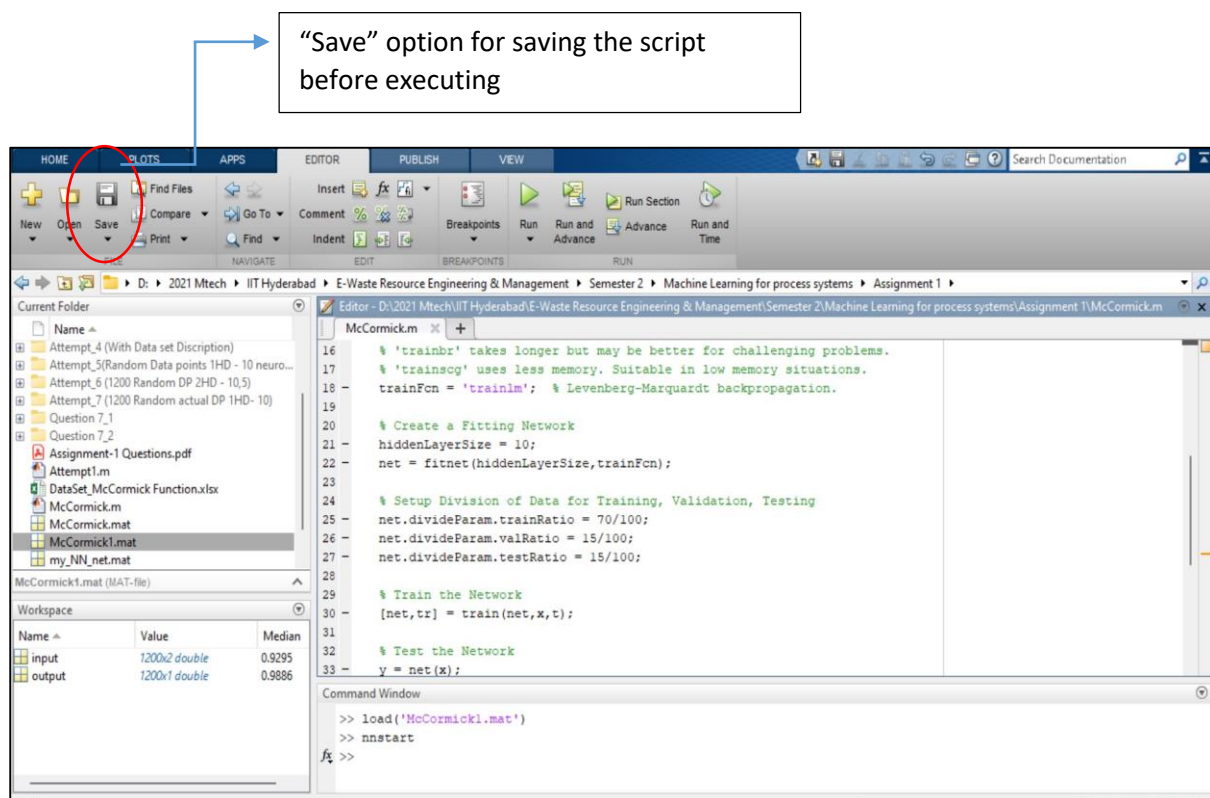
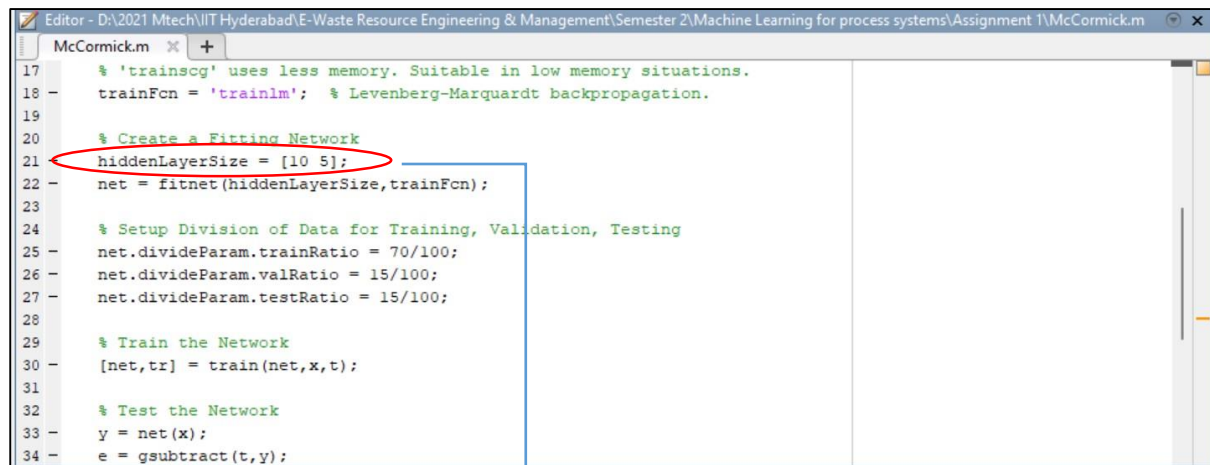


Figure 15: Generated Simple Script saved

Step 13: Now after saving the script, it can be modified according to the requirement. Here in this step, the hidden layers are increased from one layer to 2 layers just by changing a line in the script (i.e., `hiddenlayersize90 = [10 5]` replacing `[10]` which was set in Step 6 – Fig 8).

By doing so (see Fig 16) a new pop up similar to Fig 10 shows up, but it now has results and respective plots which are completely different from the previous iteration.

This elucidates us that the change in the architecture of the NN for the same data set can result in different outcomes. The best of the outcomes should be chosen after meticulous modification and interpretation of all results that are obtained after training each.

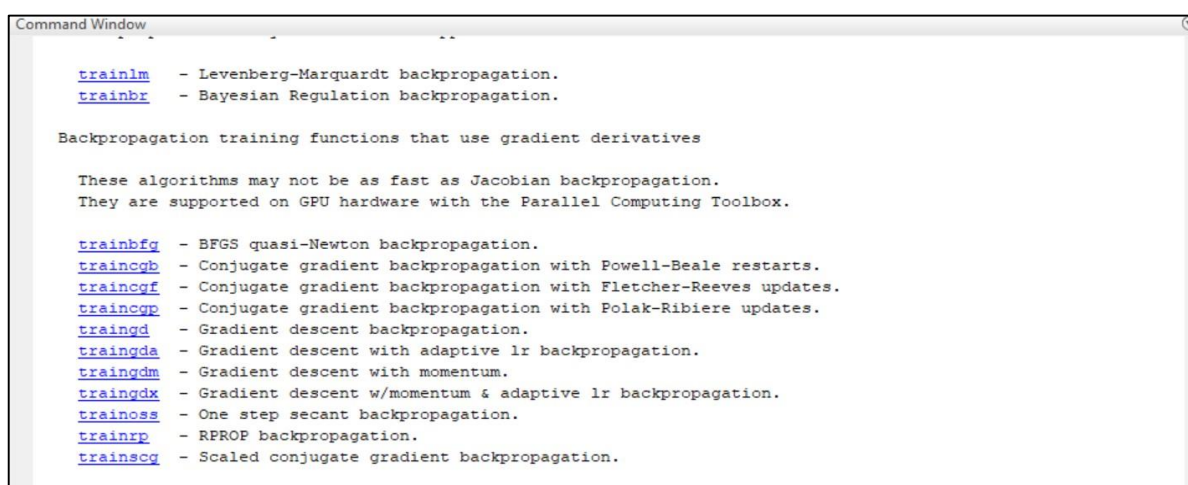


```
17 % 'trainscg' uses less memory. Suitable in low memory situations.
18 trainFcn = 'trainlm'; % Levenberg-Marquardt backpropagation.
19
20 % Create a Fitting Network
21 hiddenLayerSize = [10 5];
22 net = fitnet(hiddenLayerSize, trainFcn);
23
24 % Setup Division of Data for Training, Validation, Testing
25 net.divideParam.trainRatio = 70/100;
26 net.divideParam.valRatio = 15/100;
27 net.divideParam.testRatio = 15/100;
28
29 % Train the Network
30 [net, tr] = train(net, x, t);
31
32 % Test the Network
33 y = net(x);
34 e = gsubtract(t, y);
```

Figure 16: Hidden Layer Modification

Here is where the hidden layers can be added to the network, hence resulting with different outcomes

Step 14: Now to incorporate different algorithms for the same architecture & data set, one need to know what are the different algorithms that are available in the library. So for that, by entering “Help nntrain” in the command window, it provides us the list of algorithms and its basic description against each (as shown in Fig 17).



```
Command Window

trainlm - Levenberg-Marquardt backpropagation.
trainbr - Bayesian Regularization backpropagation.

Backpropagation training functions that use gradient derivatives

These algorithms may not be as fast as Jacobian backpropagation.
They are supported on GPU hardware with the Parallel Computing Toolbox.

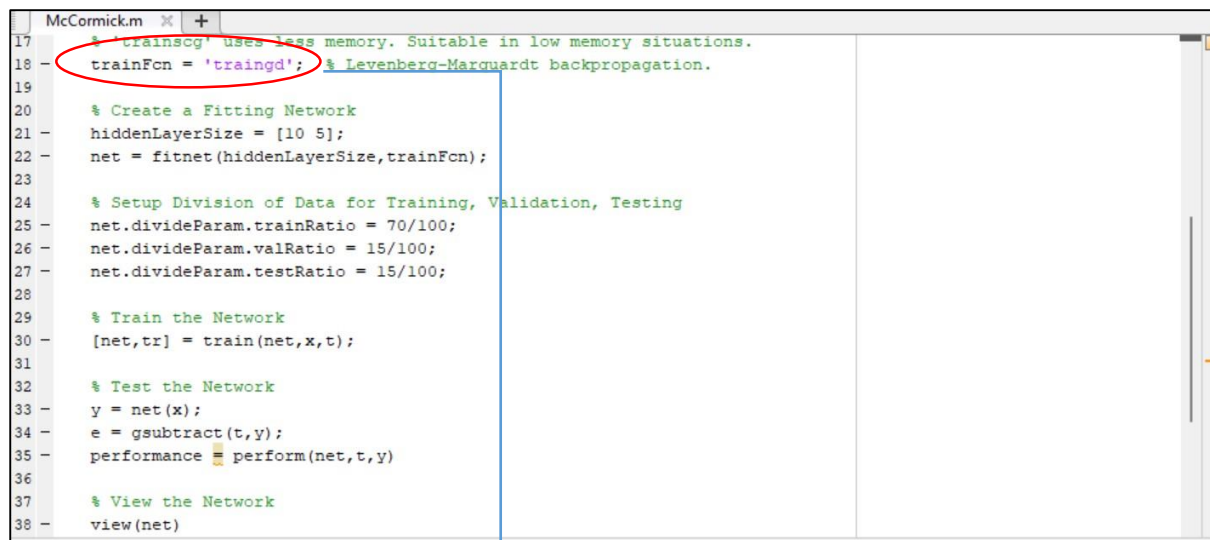
trainbfg - BFGS quasi-Newton backpropagation.
traincgb - Conjugate gradient backpropagation with Powell-Beale restarts.
traincgf - Conjugate gradient backpropagation with Fletcher-Reeves updates.
traincgp - Conjugate gradient backpropagation with Polak-Ribiere updates.
traingd - Gradient descent backpropagation.
traingda - Gradient descent with adaptive lr backpropagation.
traingdm - Gradient descent with momentum.
traingdx - Gradient descent w/momentum & adaptive lr backpropagation.
trainoss - One step secant backpropagation.
trainrp - RPROP backpropagation.
trainscg - Scaled conjugate gradient backpropagation.
```

Figure 17: Different Training Algorithms

Step 15: After choosing one among the list (see Fig 17), already integrated algorithm is replaced by the new one. This is done simply by calling the new algorithm as shown in Fig 18.

After making the replacement the script is run and we again get a pop- up same as Figure 10, but with different plots and parameters.

Other modification that can be done here is, we can change the data division percentage of training data which was not the case in step 5 (Fig 7). By doing so, it may also affect the results however with a low chance of improving the efficiency of the model.



```
McCormick.m X +
17 % 'trainlm' uses less memory. Suitable in low memory situations.
18 - trainFcn = 'traingd'; % Levenberg-Marquardt backpropagation.
19
20 % Create a Fitting Network
21 - hiddenLayerSize = [10 5];
22 - net = fitnet(hiddenLayerSize,trainFcn);
23
24 % Setup Division of Data for Training, Validation, Testing
25 - net.divideParam.trainRatio = 70/100;
26 - net.divideParam.valRatio = 15/100;
27 - net.divideParam.testRatio = 15/100;
28
29 % Train the Network
30 - [net,tr] = train(net,x,t);
31
32 % Test the Network
33 - y = net(x);
34 - e = gsubtract(t,y);
35 - performance = perform(net,t,y)
36
37 % View the Network
38 - view(net)
```

Figure 18: Replacement of New Algorithm

Here the old one “trainlm” (Levenberg-Marquardt) is replaced by “traingd” (Gradient descent backpropagation).

Playing with these hyper parameters helps in building an efficient NN and also enables the user to compare different algorithms for the same data set. “Performance value generated from the script (as shown in figure 19) is a key parameter in this comparison which alludes the average MSE of targeted and actual output.



```
Command Window
>> load('McCormick1.mat')
>> nnstart
>> McCormick

performance =
7.1524e-09
```

Figure 19: Performance value

Here the performance value is in the order of e-09, which means that the difference between the actual and targeted output is very less, hence a good model.

2. Perform an analysis by varying number of hidden nodes, training algorithms (at least five from the list) and sample size for training. Include all possible inferences, observations and comparisons in the report with relevant figures.

Hidden node variation may change the result and the convergence drastically. So here two architectures are considered i.e., [5 3] and [10 5] alongwith different models as follows:

- (i) LM (Levenberg-Marquardt backpropagation)
- (ii) GDM (Gradient Descent Back propagation with momentum)
- (iii) GD (Gradient Descent Back propagation)
- (iv) CGB (Conjugate gradient backpropagation with Powell-Beale restarts)
- (v) BR (Bayesian Regulation backpropagation)
- (vi) B (Batch training with weight & bias learning rules)

All these algorithms vary in their approach, for example LM following jacobian matrix and while the GDM doesn't. They are activated accordingly when called in the script with respective syntax. These algorithms are compared one over the other in the similar way as that of Question 5:

- a) As from the Figure E it is evident that **algorithms GD, GDM & B doesn't converge in terms of epochs with both the architecture. Whereas LM, BR & CGB show convergence** at both the cases ([5 3] & [10 5]).

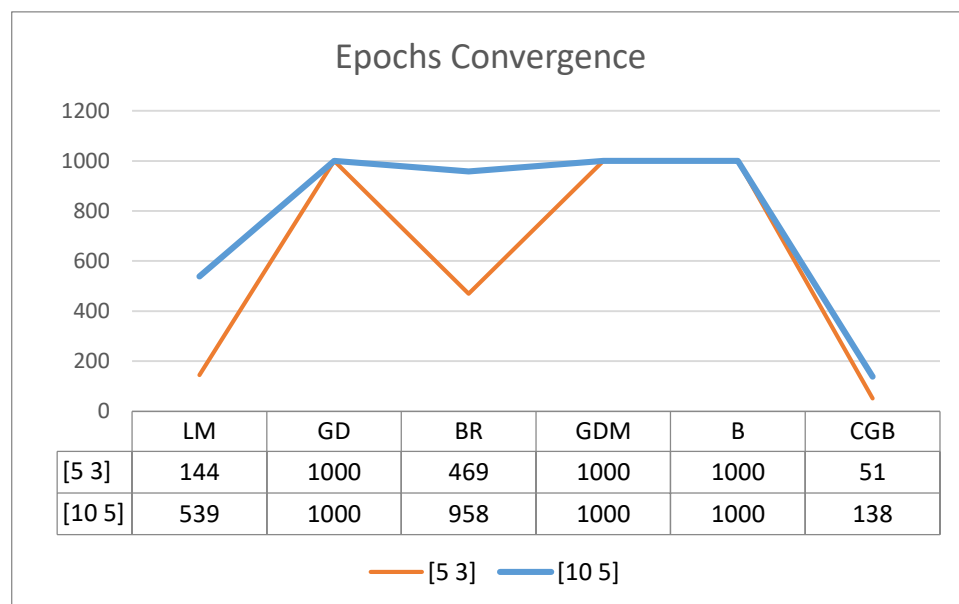


Figure E: Epoch convergence of different algorithms

From this convergence criterion we can filter out the best algorithms and can be further deeply analysed. Here as for the given sample data and different hidden nodes, LM BR & CGB seems to give prompt results comparatively.

- b) With each algorithm, prediction was carried out with fresh 100 unseen data on [5 3] & [10 5] and the prediction performance of those are recorded and compared with performance as shown in Figure F & Figure G.

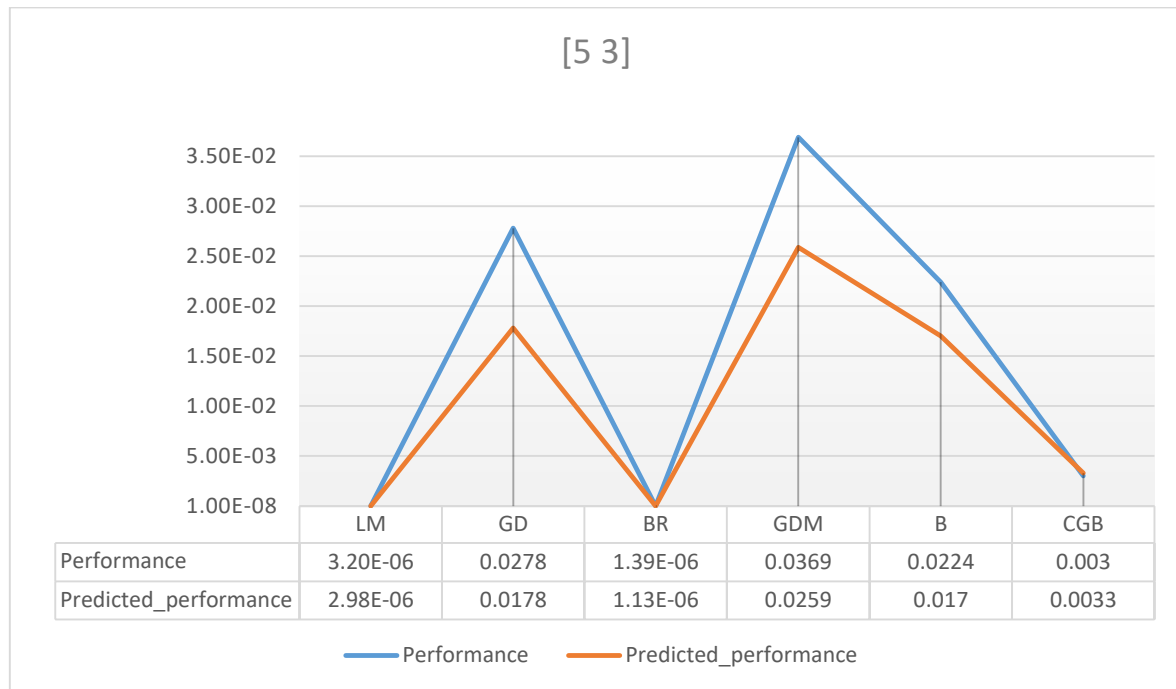


Figure F: Comparison among algorithm's Performance & Predicted performance [5 3]

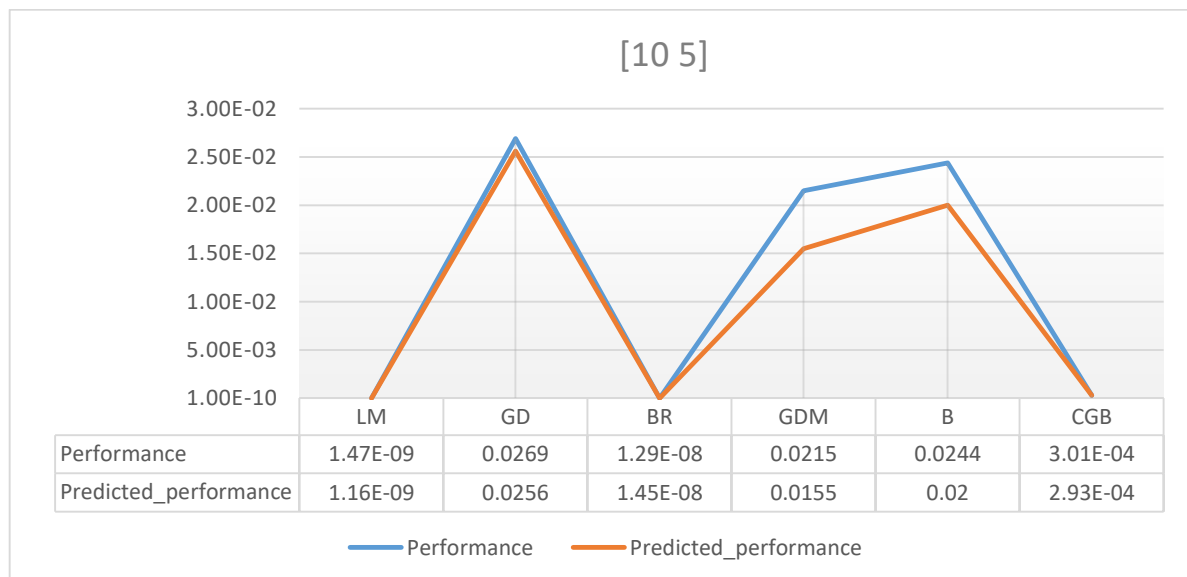


Figure G: Comparison among algorithm's Performance & Predicted performance [10 5]

It can be interpreted that – **LM, BR & CGB have minimum difference between their performance and predicted performance in both the architectures when compared to GD, GDM & B.** This information tells us the former set of algorithms (LM, BR & CGB) fit the samples more efficiently than the later (GD, GDM & B).

- c) After kind of sorting out the suitable algorithm, Regression value which denotes how perfectly the “fit” is - should also be taken into account for comparison which further filters out the algorithms (see Fig H).

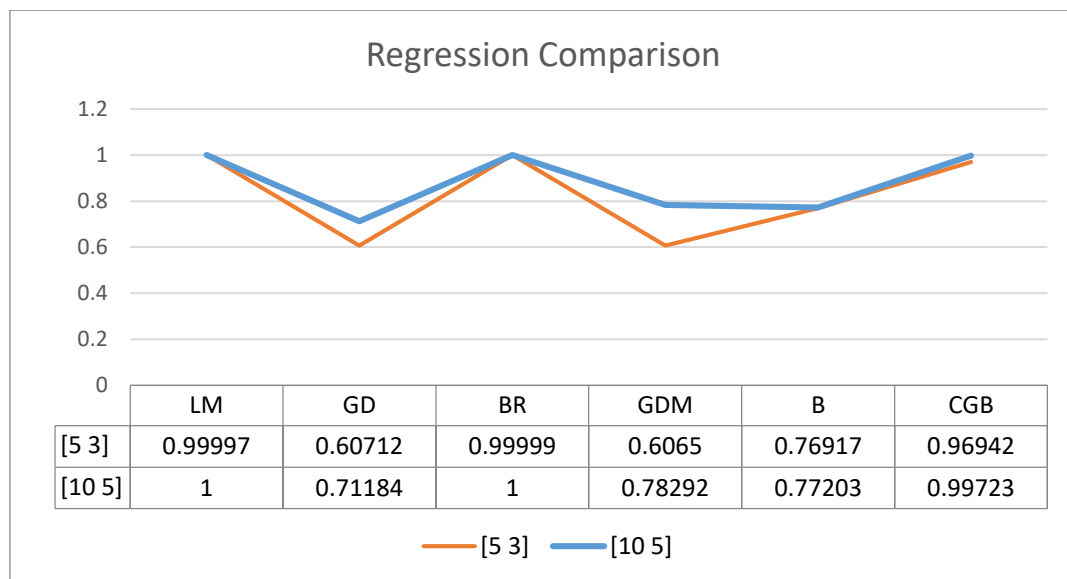


Figure H: Regression comparison among algorithms

As seen here, **R value doesn't vary a lot for LM, BR, B & CGB when applied for different architecture compared to the gradient descent methods (GD & GDM)**. However, algorithm “B” did not give satisfactory results for other parameters.

- d) There are many other inferences that can be drawn from the results which are obtained by applying different algorithms. One such is that, error histogram which is generated after each run captures the error and also shows us how many datasets among training, validation & testing fall in a particular range of error with bars. This bar plots also give hand to the user for better analysis within the dataset.

Concluding the comparison among the 6 different algorithms and with the arguments that were put forth it is clear that for the dataset given, **LM & BR algorithms give their best performance**.

These two algorithms when they were iterated more times displays the similar trend in results i.e., **low MSE, best regression and also performs with least deviation during prediction of fresh inputs**. attributes make these 2 algorithms outstand from other algorithms (GD, GDM, B & CGB).