ARTIFICIAL INTELLIGENCE PRACTICAL FILE

SUBMITTED BY: MANAN MADAN
ROLL NUMBER: 2018UIC3087
Branch: Instrumentation and Control Engineering

INDEX

Practical 1: To implement the Bayesian Regularization algorithm

Theory:

Bayesian regularization minimizes a linear combination of squared errors and weights. It also modifies the linear combination so that at the end of training the resulting network has good generalization qualities.
This Bayesian regularization takes place within the Levenberg-Marquardt algorithm. Backpropagation is used to calculate the Jacobian jX of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to Levenberg-Marquardt,
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
where E is all errors and I is the identity matrix.

The adaptive value mu is increased by mu_inc until the change shown above results in a reduced performance value. The change is then made to the network, and mu is decreased by mu_dec.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- mu exceeds mu_max.

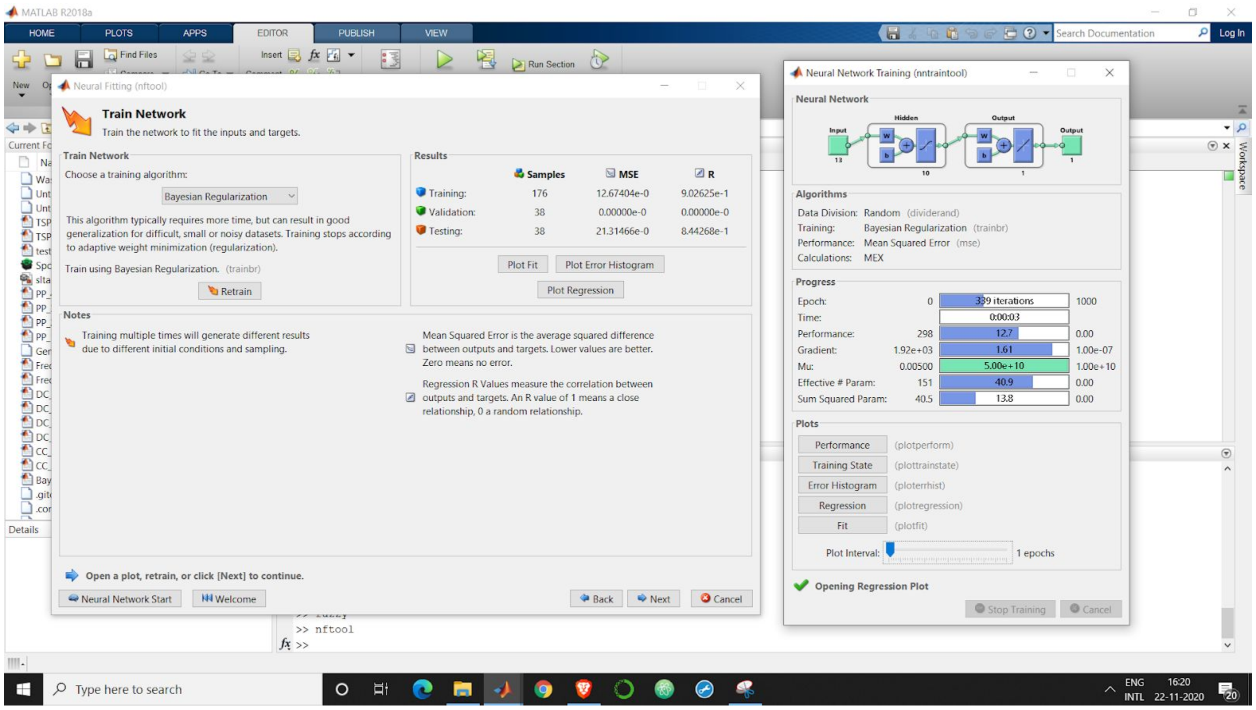
We have performed Bayesian Regularization using the Neural Fitting (NFTOOL) Tool of Matlab. Various sample data tests provided by NFTOOL for were considered for training, testing and output before finally using the following data:
Inputs: 'bodyfatInputs' is a 13x252 matrix, representing static data: 252 samples of 13 elements.
Targets: 'bodyfatTargets' is a 1x252 matrix, representing static data: 252 samples of 1 element.

Three types of Data Samples are:

- **Training**: These are presented to the network during training, and the network is adjusted according to its error. (70%)
- **Validation**: These are used to measure network generalization, and to halt training when generalization stops improving. (15%)
- **Testing**: These have no effect on training and so provide an independent measure of network performance during and after training. (15%)

The algorithm used to train the neural network: Bayesian Regularization algorithm

This algorithm typically requires more time but can result in good generalization for difficult, small, or noisy datasets. Training stops according to adaptive weight minimization (regularization).


MATLAB R2018a

Neural Fitting (nftool)

Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm: Bayesian Regularization

This algorithm typically requires more time, but can result in good generalization for difficult, small or noisy datasets. Training stops according to adaptive weight minimization (regularization).

Train using Bayesian Regularization. (trainbr)

Results

	Samples	MSE	R
Training:	176	12.67404e-0	9.02625e-1
Validation:	38	0.00000e-0	0.00000e-0
Testing:	38	21.31466e-0	8.44268e-1

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Open a plot, retrain, or click [Next] to continue.

```
>> nftool
>>
```

Neural Network Training (nntraintool)

Algorithms

Data Division:	Random (dividerand)
Training:	Bayesian Regularization (trainbr)
Performance:	Mean Squared Error (mse)
Calculations:	MEX

Progress

Epoch:	0	339 iterations	1000
Time:		0:00:03	
Performance:	298	12.7	0.00
Gradient:	1.92e+03	1.61	1.00e-07
Mu:	0.00500	5.00e+10	1.00e+10
Effective # Param:	151	40.9	0.00
Sum Squared Param:	40.5	13.8	0.00

Plot Interval: 1 epochs

Opening Regression Plot

MATLAB Function's Code:

```
function [Y,Xf,Af] = myNeuralNetworkFunction(X,~,~)
%MYNEURALNETWORKFUNCTION neural network simulation function.
% Generated by Neural Network Toolbox function genFunction, 22-Nov-2020 16:24:05.
% [Y] = myNeuralNetworkFunction(X,~,~) takes these arguments:
%   X = 1xTS cell, 1 inputs over TS timesteps
%   Each X{1,ts} = 13xQ matrix, input #1 at timestep ts.
% and returns:
%   Y = 1xTS cell of 1 outputs over TS timesteps.
%   Each Y{1,ts} = 1xQ matrix, output #1 at timestep ts.
% where Q is number of samples (or series) and TS is the number of timesteps.

% ===== NEURAL NETWORK CONSTANTS =====
% Input 1
x1_step1.xoffset = [22;118.5;29.5;31.1;79.3;69.4;85;47.2;33;19.1;24.8;21;15.8];
x1_step1.gain                                                                   =
[0.0338983050847458;0.00817494379726139;0.0414507772020725;0.099502487562189;0.035
1493848857645;0.0254129606099111;0.0318979266347687;0.0498753117206983;0.12422360
2484472;0.135135135135135;0.099009900990099;0.143884892086331;0.357142857142857];
x1_step1.ymin = -1;

% Layer 1
b1                                                                              =
[0.0062506598832845587654;-0.12081030746663769249;-0.0062506598867840034231;-0.006
2506599294972426187;-0.0062506598821270012087;-0.42687246239220916211;0.229312138
80811640582;-0.0062506599528449387909;0.0062506598873213244788;0.1185745518344051
147];
IW1_1         =         [0.048829962163003580444         -0.041605297978994724983
0.0066328550177292555645      -0.091970156596653168668      -0.11050645563369204671
0.072994499412426194773       0.017823206773652622292       0.071145788307842294795
0.0078299595166819117159      -0.0086117463737760153314     -0.040997757698026611461
```

```
-0.066238365406660254 52                    -0.1106763339317666378 1;0.2168446435997303800 6
```

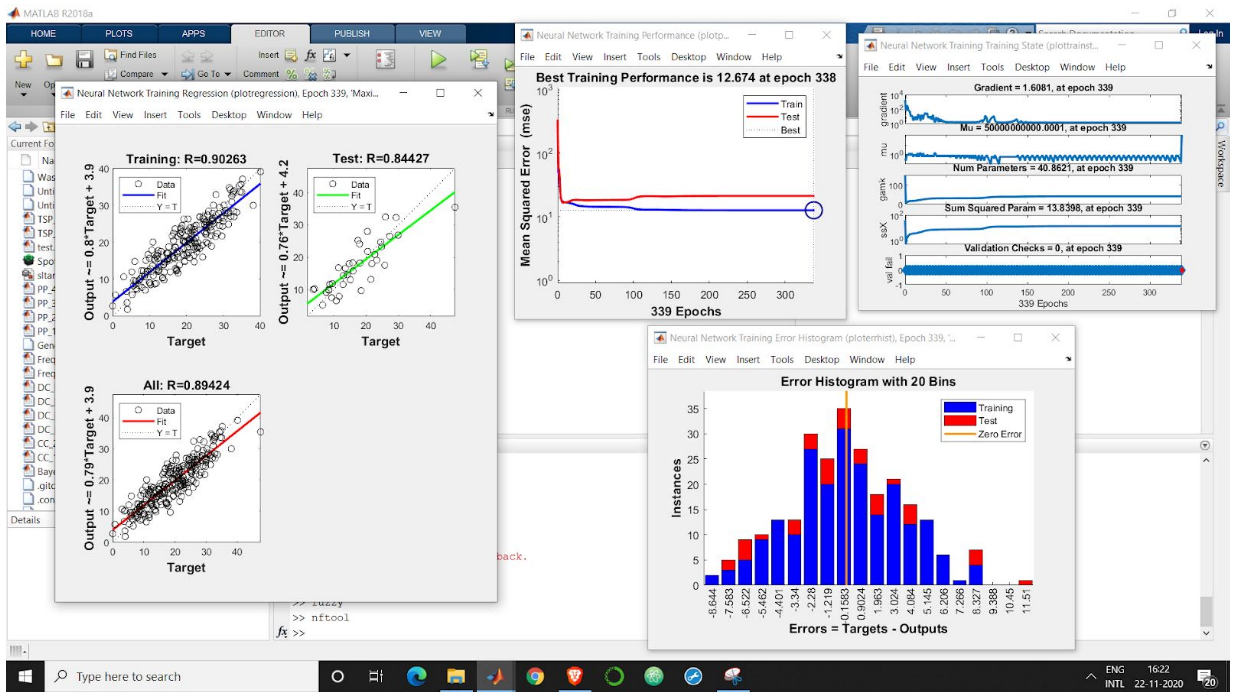
```
-0.066238365406660254 52
```

```
-0.06623836540666025452                  -0.11067633393176663781;0.21684464359973038006
0.06432813844173772275           0.10912959460785026655           0.32741557628985612505
-0.09975059612581231582 9        -0.31878039337138974751          0.40438298933189170681
-0.1217115588139587451          -0.32358664178822404978           -1.2290201287356063986
0.3358636934922413219 7                                           0.34556530255164558119
-0.66914691746396137706;-0.04882996212093391325                  0.04160529796052776091 4
-0.00663285501940327586 19       0.09197015653978163340 9         0.11050645555559850119
-0.07299449933400815882         -0.01782320674408925317 8        -0.07114578822736479513 1
-0.00782995949937673532 68       0.00861174637322996081 03        0.04099775768554032862
0.06623836536480219583 4                  0.11067633385150774106;-0.04882996160676649627 4
0.04160529773441067752 4        -0.00663285503978539459 77        0.09197015584432831347 7
0.11050645460104890905          -0.07299449837631639959          -0.01782320638322856115 6
-0.07114578724443018642 5       -0.00782995928835092565 3        0.00861174636619878482 17
0.04099775753258368821 1                                          0.06623836485307928179 1
0.11067633287066637 46;-0.04882996217693263 28                    0.04160529798511050647 6
-0.00663285501717439905 66       0.09197015661548263443 2         0.11050645565954878013
-0.07299449943838648424 7       -0.01782320678344020276 1        -0.07114578833448513550 7
-0.00782995952240970322 09       0.00861174637395908069 98        0.04099775770216160936 1
0.06623836542051905751 4                   0.11067633395833870 18;-0.41371680322000414787
-0.19773958473260283553         -0.25905607403098029895           -0.70593575958058762954
-0.31815561343954679163         -0.73160395997465887952           -0.24296074766139841294
0.10488627295411510898           0.15341287710943077305           0.62127127900295764373
-0.20075434328566735265                                           0.34073859765141606415
0.13745877817799495579;0.57214748060286302334                    -0.01062188591808201076 9
0.23112386146109820118           0.59201109380141903049           0.25068953718605380132
-0.34003190386730591799          -0.13420676607143949832          -0.46103840985366634 22
-0.03341376944991964609 3         0.422191538488255802            -0.33583460116646951521
-0.63113460907222451723                    0.40630162707839406755;-0.04882996132533315436 5
0.04160529761050781733          -0.00663285505091473683 72        0.09197015546352518833 3
0.11050645407852283109          -0.07299449785234428389 5         -0.01782320618585379185
-0.07114578670662125015 1       -0.00782995917299968667 57       0.00861174636222986240 87
0.04099775744874521038 5                                          0.06623836457292865687
0.11067633233380565205;0.04882996211450191942 3                   -0.04160529795770081390 4
0.00663285501965967493 05       -0.09197015653108435440 1         -0.11050645554365928769
0.07299449932202614621 5         0.01782320673957452084           0.07114578821506740979 9
0.00782995949673621557 95       -0.00861174637314523171 16       -0.04099775768362763639 4
-0.06623836535840270378 6                  -0.11067633383923942969;-0.62314803114242933724
0.30992394050977178921           0.19444215059712849358           -0.03666395308881659903 5
0.09548092774820593486 9         0.06003266899131421557 9         0.59284562133666518502
0.58295588063990433358           0.37097102314701413395           0.10989934764280584467
0.07285727632373190099 1 -0.44763012124425916038 0.27967873137913035197];

% Layer 2
b2 = 0.04903782668744958994 6;
LW2_1   =   [0.24255642087527148898   -1.1532071158921515242   -0.24255642068800711 83
-0.24255641840101571649          -0.24255642093726673125           -1.1484518424494771782
-1.0644639632157661957           -0.24255641714978101731           0.24255642065939123087
-0.79711023611935094557];

% Output 1
y1_step1.ymin = -1;
y1_step1.gain = 0.0421052631578947;
```

```
y1_step1.xoffset = 0;

% ===== SIMULATION ========
% Format Input Arguments
isCellX = iscell(X);
if ~isCellX
   X = {X};
end
% Dimensions
TS = size(X,2); % timesteps
if ~isempty(X)
   Q = size(X{1},2); % samples/series
else
   Q = 0;
end
% Allocate Outputs
Y = cell(1,TS);
% Time loop
for ts=1:TS

   % Input 1
   Xp1 = mapminmax_apply(X{1,ts},x1_step1);

   % Layer 1
   a1 = tansig_apply(repmat(b1,1,Q) + IW1_1*Xp1);

   % Layer 2
   a2 = repmat(b2,1,Q) + LW2_1*a1;

   % Output 1
   Y{1,ts} = mapminmax_reverse(a2,y1_step1);
end

% Final Delay States
Xf = cell(1,0);
Af = cell(2,0);
% Format Output Arguments
if ~isCellX
   Y = cell2mat(Y);
end
end

% ===== MODULE FUNCTIONS ========
% Map Minimum and Maximum Input Processing Function
function y = mapminmax_apply(x,settings)
y = bsxfun(@minus,x,settings.xoffset);
y = bsxfun(@times,y,settings.gain);
y = bsxfun(@plus,y,settings.ymin);
end
% Sigmoid Symmetric Transfer Function
function a = tansig_apply(n,~)
a = 2 ./ (1 + exp(-2*n)) - 1;
```

```
end
% Map Minimum and Maximum Output Reverse-Processing Function
function x = mapminmax_reverse(y,settings)
x = bsxfun(@minus,y,settings.ymin);
x = bsxfun(@rdivide,x,settings.gain);
x = bsxfun(@plus,x,settings.xoffset);
end
```

Output Plots:



MATLAB R2018a
Neural Network Training Regression (plotregression), Epoch 339, 'Maxi...
Training: R=0.90263
Output ~= 0.8*Target + 3.9
Test: R=0.84427
Output ~= 0.76*Target + 4.2
All: R=0.89424
Output ~= 0.79*Target + 3.9
Data
Fit
Y = T
Target
Neural Network Training Performance (plotp...
Best Training Performance is 12.674 at epoch 338
Mean Squared Error (mse)
Train
Test
Best
339 Epochs
Neural Network Training Training State (plottrainst...
Gradient = 1.6081, at epoch 339
Mu = 50000000000.0001, at epoch 339
Num Parameters = 40.8621, at epoch 339
Sum Squared Param = 13.8398, at epoch 339
Validation Checks = 0, at epoch 339
339 Epochs
Neural Network Training Error Histogram (ploterrhist), Epoch 339, '...
Error Histogram with 20 Bins
Training
Test
Zero Error
Instances
Errors = Targets - Outputs

```
>> fuzzy
>> nftool
>>
```

Practical 2: To implement the Levenberg Marquardt Algorithm

Theory: The Levenberg-Marquardt algorithm, also known as the damped least-squares method, has been designed to work specifically with loss functions, which take the form of a sum of squared errors. It works without computing the exact Hessian matrix. Instead, it works with the gradient vector and the Jacobian matrix.

Consider a loss function which can be expressed as a sum of squared errors of the form

$$f = \sum_{i=1}^{m} e_i^2$$

Here m is the number of instances in the data set.

We can define the Jacobian matrix of the loss function as that containing the derivatives of the errors concerning the parameters,

$$\mathbf{J}_{i,j} = \frac{\partial e_i}{\partial \mathbf{w}_j},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

Where m is the number of instances in the data set, and n is the number of parameters in the neural network. Note that the size of the Jacobian matrix is $m \cdot n$

The gradient vector of the loss function can be computed as:

$$\nabla f = 2\mathbf{J}^T \cdot \mathbf{e}$$

Here e is the vector of all error terms.

Finally, we can approximate the Hessian matrix with the following expression.

$$\mathbf{H}f \approx 2\mathbf{J}^T \cdot \mathbf{J} + \lambda \mathbf{I}$$

Where λ is a damping factor that ensures the positiveness of the Hessian and I is the identity matrix.

The next expression defines the parameters improvement process with the Levenberg-Marquardt algorithm

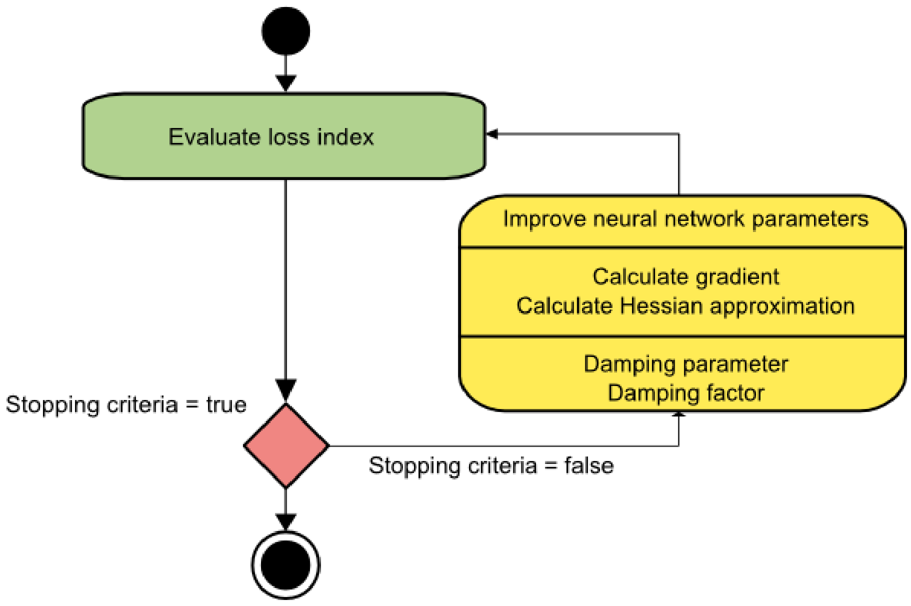
$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - (\mathbf{J}^{(i)T} \cdot \mathbf{J}^{(i)} + \lambda^{(i)}\mathbf{I})^{-1} \cdot (2\mathbf{J}^{(i)T} \cdot \mathbf{e}^{(i)}),$$

for i = 0,1...

When the damping parameter λ is zero, this is just Newton's method, using the approximate Hessian matrix. On the other hand, when λ is large, this becomes gradient descent with a small training rate.

The parameter λ is initialized to be large so that the first updates are small steps in the gradient descent direction. If any iteration happens to result in a fail, then λ is increased by some factor. Otherwise, as the loss decreases, λ is decreased so that the Levenberg-Marquardt algorithm approaches the Newton method. This process typically accelerates the convergence to the minimum.

The picture below represents a state diagram for the training process of a neural network with the Levenberg-Marquardt algorithm. The first step is to calculate the loss, the gradient, and the Hessian approximation. Then the damping parameter is adjusted to reduce the loss at each iteration.



As we have seen, the Levenberg-Marquardt algorithm is a method tailored for functions of the type sum-of-squared-error. That makes it to be very fast when training neural networks measured on that kind of error.

However, this algorithm has some drawbacks. The first one is that it cannot be applied to functions such as the root mean squared error or the cross-entropy error. Also, for big data sets and neural networks, the Jacobian matrix becomes enormous, and therefore it requires much memory. Therefore, the Levenberg-Marquardt algorithm is not recommended when we have big data sets or neural networks.

We have performed Bayesian Regularization using the Neural Fitting (NFTOOL) Tool of Matlab. Various sample data tests provided by NFTOOL for were considered for training, testing and output before finally using the following data:
Inputs: 'bodyfatInputs' is a 13x252 matrix, representing static data: 252 samples of 13 elements.
Targets: 'bodyfatTargets' is a 1x252 matrix, representing static data: 252 samples of 1 element.

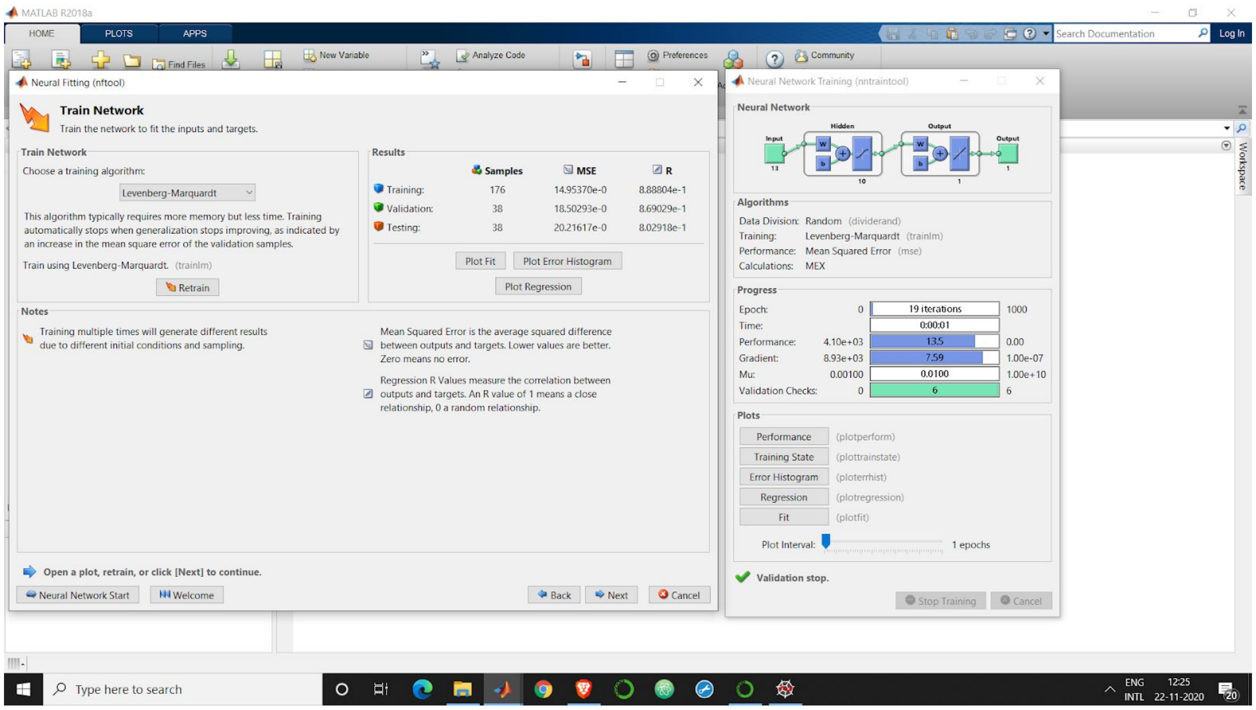
Three types of Data Samples are:

- **Training**: These are presented to the network during training, and the network is adjusted according to its error. (70%)

- **Validation**: These are used to measure network generalization, and to halt training when generalization stops improving. (15%)
- **Testing**: These have no effect on training and so provide an independent measure of network performance during and after training. (15%)

The algorithm used to train the neural network: Levenberg Marquardt Algorithm.

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.



Neural Fitting (nftool)

Train Network

Train the network to fit the inputs and targets.

Train Network

Choose a training algorithm:

Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt. (trainlm)

Retrain

Results

	Samples	MSE	R
Training:	176	14.95370e-0	8.88804e-1
Validation:	38	18.50293e-0	8.69029e-1
Testing:	38	20.21617e-0	8.02918e-1

Plot Fit · Plot Error Histogram · Plot Regression

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Open a plot, retrain, or click [Next] to continue.

Neural Network Start · Welcome · Back · Next · Cancel

Neural Network Training (nntraintool)

Neural Network

Algorithms

Data Division:	Random (dividerand)
Training:	Levenberg-Marquardt (trainlm)
Performance:	Mean Squared Error (mse)
Calculations:	MEX

Progress

Epoch:	0	19 iterations	1000
Time:		0:00:01	
Performance:	4.10e+03	13.5	0.00
Gradient:	8.93e+03	7.59	1.00e-07
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

Plots

Performance	(plotperform)
Training State	(plottrainstate)
Error Histogram	(ploterrhist)
Regression	(plotregression)
Fit	(plotfit)

Plot Interval: 1 epochs

Validation stop.

Stop Training · Cancel

Code for the algorithm:

```
function [Y,Xf,Af] = myNeuralNetworkFunction(X,~,~)
%MYNEURALNETWORKFUNCTION neural network simulation function.
% Generated by Neural Network Toolbox function genFunction, 15-Nov-2020 12:31:11.
% [Y] = myNeuralNetworkFunction(X,~,~) takes these arguments:
%   X = 1xTS cell, 1 inputs over TS timesteps
%   Each X{1,ts} = 13xQ matrix, input #1 at timestep ts.
% and returns:
%   Y = 1xTS cell of 1 outputs over TS timesteps.
%   Each Y{1,ts} = 1xQ matrix, output #1 at timestep ts.
% where Q is number of samples (or series) and TS is the number of timesteps.
```

% ===== NEURAL NETWORK CONSTANTS =====

```
% Input 1
x1_step1.xoffset = [22;118.5;29.5;31.1;79.3;69.4;85;47.2;33;19.1;24.8;21;15.8];
x1_step1.gain =
[0.0338983050847458;0.00817494379726139;0.0414507772020725;0.0995024875621890;0.035
1493848857645;0.0254129606099111;0.0318979266347687;0.0498753117206983;0.12422360
```

```
2484472;0.135135135135135;0.099009900990099;0.143884892086331;0.357142857142857];
x1_step1.ymin = -1;

% Layer 1
b1                                                                               =
[31.374291320158160801;-36.601064902228095832;22.299398075243608 73;-56.35100602780
1155085;0.92877561556889465244;0.0095786886336579613044;-7.0823952882671097342;-6
1.154673393296391737;-56.334821975905391866;207.60706933888013737];
IW1_1    =   [137.52796043743950349    -39.097869287764950741    -58.197942842220754756
84.295004218244088179            41.028065519186391441            29.814983289067132688
-4.4468414411064625114           -53.507349231263532374           -164.47314164823069405
-77.743650243569817349            60.683298123035896765            22.254077919338815406
30.273078013382122009;64.350707114196822545                        106.21979383391871465
203.32918140290391307            -34.010455262266773957           -175.73306366999977968
60.252752604618429189            15.094771239818523867            101.01418192664837647
-145.07037784337052244           107.82336494654241221            81.320309199462514016
-14.450405313811945263                      -17.133083116901921983;5.9395121547018 4124
-0.63025806568661746354          -13.570211674087159892           -0.67354656191914585861
14.916738506554526822            -13.369269718683545634            1.3327110381411948481
34.212670720376451072            3.5883140426116764132            -14.803457089912051003
10.149129604961876439                                             -14.600931635125927954
-3.8349622617090881604;23.162277364495366783                       52.819815933936517638
-15.778964335185669654           -6.4622052636600306741           -25.307126819155218556
8.3122164629255159696             11.026611624130479683             17.596259227133117 52
21.644737060536357376            12.257733017008096255            -3.0025894171318636694
5.2196734238975412978                  -20.345579524325742682;-0.081535507925392547435
-0.10040829129094103189         -0.011463436186592373955         -0.61968903564627308977
-0.66857727068528649 55          2.9120143435213776684            -0.25015755219219953931
-0.19951491056274658908           0.38530632933518182 97            0.14993384550553001677
0.26364023271670544712                                          -0.00011316944561191172314
-0.36534262972269160308;20.798139016832916326                     -100.13017995407349758
-71.543592211560664396            -7.3891122778749309674           172.96227892238960067
-138.51103406022349418            118.76014205162147164            49.008938655566190334
-105.09118388494374585            -18.595175509518142576            -17.554926364580389 02
55.631992855041929147                    -96.511182575734324018;70.470938711600240367
14.430042662929713 16             52.198944413524138497            38.323461052926738546
76.221536835624817741            -185.27357727621674144            65.537816582117230269
206.83846036121406087            -159.7633751443148924             9.4484472973323718747
-80.574208478057258276                                            19.735956572960297706
18.284256107370580935;-85.045874604102962735                       39.549004792912384687
5.9697683388738573952            46.991103281204367192            -125.06181169996055758
-10.501868298517287315            48.210439080040167426            -54.667355526154942424
-43.904829006378108147           -79.648016228785493809            140.37533452860219541
-49.945251968565621326                   263.80319900477115 88;40.508870320128366416
-137.94572105548976992           -48.109185379457443332            -79.300516325004608 12
65.085408684004178781            24.902744510013885559            -58.534564939299926323
-48.884645392779390249            99.125926068945744873            -39.476606445266988032
-112.92773919302921115                                            -81.780481774744671952
92.498873035146232269;39.438795743368196156                        319.38672832270788149
-70.429541539950548668            46.287781455011170806            -67.948407155688656189
-72.669760206510986222             -71.042195298792847 78           145.62724010458421731
-54.663376385921807 81            104.13631770106428576             10.377139962542504037
```

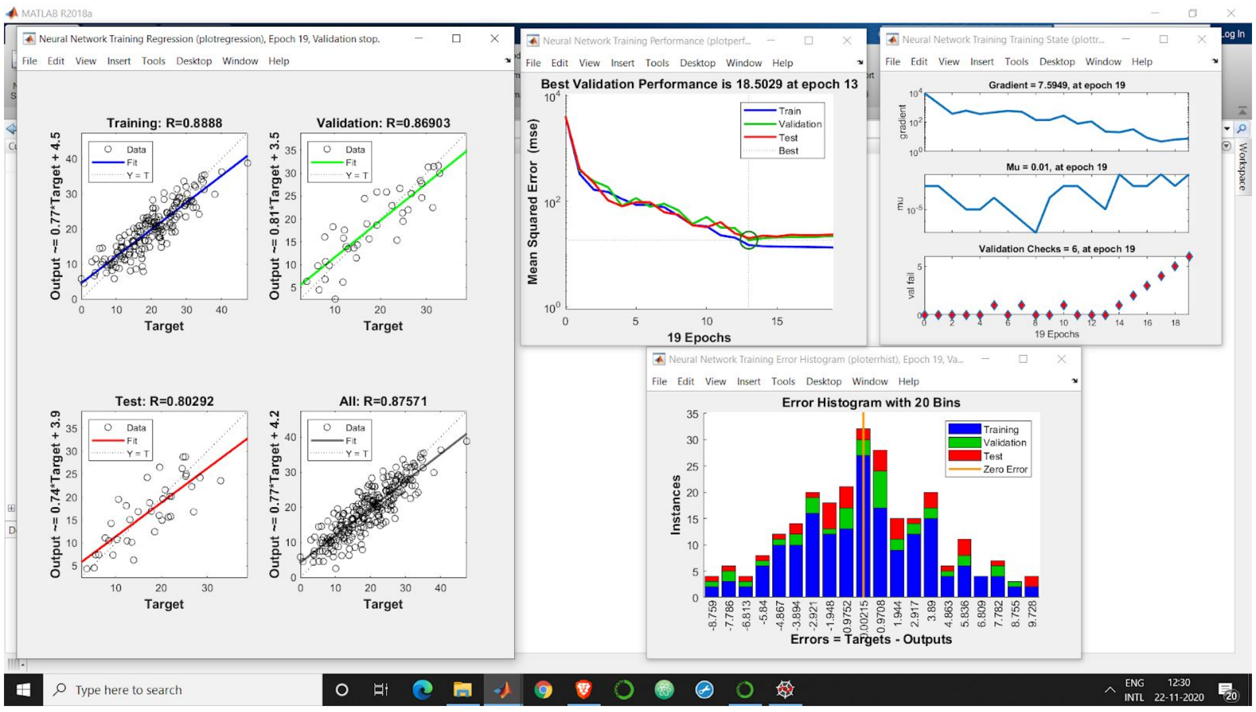
```
-42.904387823795055112 -7.1469164835430500915];

% Layer 2
b2 = -0.046943932189109409403;
LW2_1          =          [0.048627508903720098599          -0.032700264797032342623
0.015435559510624943108         0.22687926371430572337         0.74050134144269674774
0.028684271273009348535        0.058058429054067224595        0.0096303402535575441 39
-0.040778274550346715888 0.025571254221667508861];

% Output 1
y1_step1.ymin = -1;
y1_step1.gain = 0.0421052631578947;
y1_step1.xoffset = 0;
```

% ===== SIMULATION ========

```
% Format Input Arguments
isCellX = iscell(X);
if ~isCellX
   X = {X};
end
% Dimensions
TS = size(X,2); % timesteps
if ~isempty(X)
   Q = size(X{1},2); % samples/series
else
   Q = 0;
end
% Allocate Outputs
Y = cell(1,TS);

% Time loop
for ts=1:TS

   % Input 1
   Xp1 = mapminmax_apply(X{1,ts},x1_step1);

   % Layer 1
   a1 = tansig_apply(repmat(b1,1,Q) + IW1_1*Xp1);

   % Layer 2
   a2 = repmat(b2,1,Q) + LW2_1*a1;

   % Output 1
   Y{1,ts} = mapminmax_reverse(a2,y1_step1);
end
% Final Delay States
Xf = cell(1,0);
Af = cell(2,0);
% Format Output Arguments
if ~isCellX
   Y = cell2mat(Y);
```

```
end

% ===== MODULE FUNCTIONS ========
% Map Minimum and Maximum Input Processing Function
function y = mapminmax_apply(x,settings)
y = bsxfun(@minus,x,settings.xoffset);
y = bsxfun(@times,y,settings.gain);
y = bsxfun(@plus,y,settings.ymin);
end

% Sigmoid Symmetric Transfer Function
function a = tansig_apply(n,~)
a = 2 ./ (1 + exp(-2*n)) - 1;
end

% Map Minimum and Maximum Output Reverse-Processing Function
function x = mapminmax_reverse(y,settings)
x = bsxfun(@minus,y,settings.ymin);
x = bsxfun(@rdivide,x,settings.gain);
x = bsxfun(@plus,x,settings.xoffset);
End
```

Output Plots:



MATLAB R2018a
Neural Network Training Regression (plotregression), Epoch 19, Validation stop.
Training: R=0.8888
Output ~= 0.77*Target + 4.5
Validation: R=0.86903
Output ~= 0.81*Target + 3.5
Test: R=0.80292
Output ~= 0.74*Target + 3.9
All: R=0.87571
Output ~= 0.77*Target + 4.2
Data
Fit
Y = T
Target
Neural Network Training Performance (plotperf...
Best Validation Performance is 18.5029 at epoch 13
Mean Squared Error (mse)
Train
Validation
Test
Best
19 Epochs
Neural Network Training Training State (plottr...
Gradient = 7.5949, at epoch 19
gradient
Mu = 0.01, at epoch 19
mu
Validation Checks = 6, at epoch 19
val fail
19 Epochs
Neural Network Training Error Histogram (ploterrhist), Epoch 19, Va...
Error Histogram with 20 Bins
Instances
Training
Validation
Test
Zero Error
Errors = Targets - Outputs
-8.759
-7.786
-6.813
-5.84
-4.867
-3.894
-2.921
-1.948
-0.9752
-0.00215
0.9708
1.944
2.917
3.89
4.863
5.836
6.809
7.782
8.755
9.728
Type here to search
ENG INTL
12:30
22-11-2020

Practical 3: To implement the Scaled Conjugate algorithm

Theory:

From an optimization point of view learning in a neural network is equivalent to minimizing a global error function, which is a multivariate function that depends on the weights in the network.

Many of the training algorithms are based on the gradient descent algorithm. Minimization is a local iterative process in which an approximation to the function, in a neighborhood of the current point in the weight space, is minimized. Most of the optimization methods used to minimize functions are based on the same strategy.

The Scaled Conjugate Gradient (SCG) algorithm denotes the quadratic approximation to the error E in a neighborhood of a point w by:

$$E_{qw}(y) = E(w) + E'(w)^T y + \frac{1}{2} y^T E''(w) y$$

In order to determine the minimum to Eqw(y)the critical points for Eqw(y) must be found. The critical points are the solution to the linear system defined by Moller in

$$E'_{qw}(y) = E''(w)\, y + E'(w) = 0$$

SCG belongs to the class of Conjugate Gradient Methods, which show superlinear convergence on most problems. By using a step size scaling mechanism SCG avoids a time consuming line-search per learning iteration, which makes the algorithm faster than other second order algorithms. And also we got better results than with other training methods and neural networks tested, as standard back-propagation and cascade neural networks.

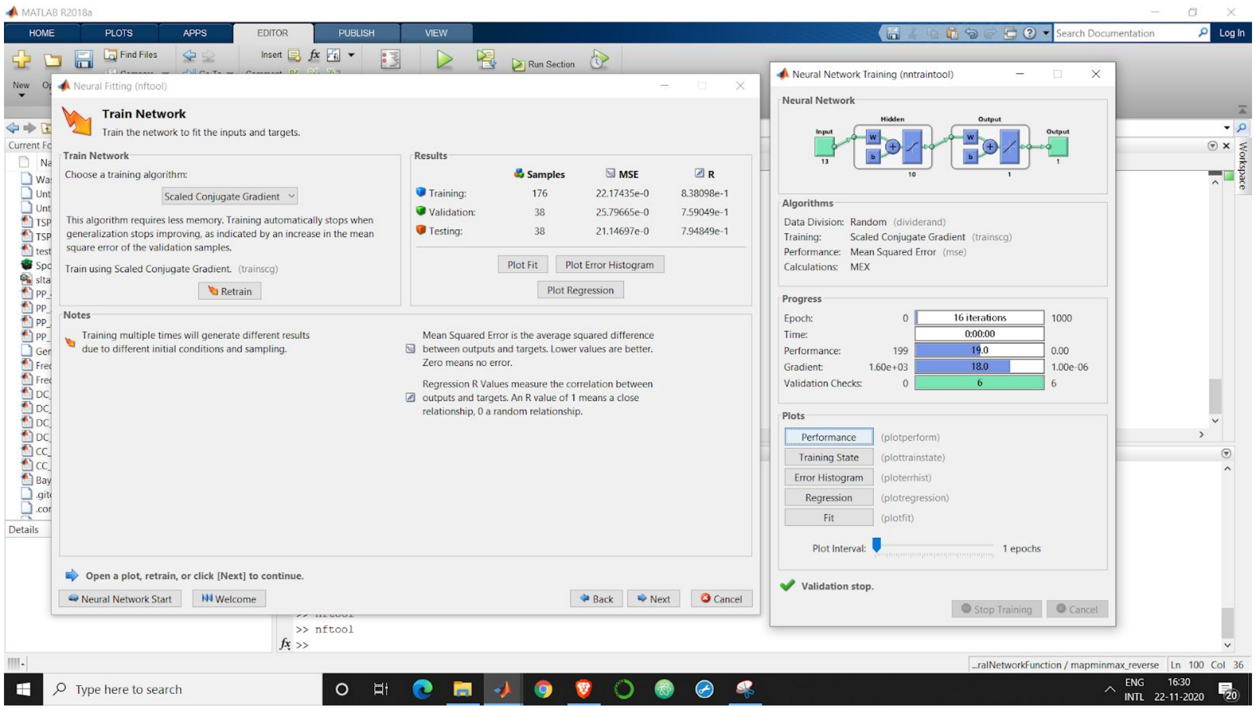
Inputs: 'bodyfatInputs' is a 13x252 matrix, representing static data: 252 samples of 13 elements.
Targets: 'bodyfatTargets' is a 1x252 matrix, representing static data: 252 samples of 1 element.

Three types of Data Samples are:

- **Training**: These are presented to the network during training, and the network is adjusted according to its error. (70%)
- **Validation**: These are used to measure network generalization, and to halt training when generalization stops improving. (15%)
- **Testing**: These have no effect on training and so provide an independent measure of network performance during and after training. (15%)

The algorithm used to train the neural network: Scaled Conjugate algorithm

This algorithm requires less memory. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.



Train Network
Train the network to fit the inputs and targets.

Train Network
Choose a training algorithm: Scaled Conjugate Gradient

This algorithm requires less memory. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Scaled Conjugate Gradient. (trainscg)

Results

	Samples	MSE	R
Training:	176	22.17435e-0	8.38098e-1
Validation:	38	25.79665e-0	7.59049e-1
Testing:	38	21.14697e-0	7.94849e-1

Notes
Training multiple times will generate different results due to different initial conditions and sampling.

Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Open a plot, retrain, or click [Next] to continue.

```
>> nftool
>>
```

Neural Network

Algorithms
Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Mean Squared Error (mse)
Calculations: MEX

Progress

Epoch:	0	16 iterations	1000
Time:		0:00:00	
Performance:	199	19.0	0.00
Gradient:	1.60e+03	18.0	1.00e-06
Validation Checks:	0	6	6

Plots
(plotperform)
(plottrainstate)
(ploterrhist)
(plotregression)
(plotfit)
Plot Interval: 1 epochs

Validation stop.

...ralNetworkFunction / mapminmax_reverse Ln 100 Col 36
ENG INTL 16:30 22-11-2020

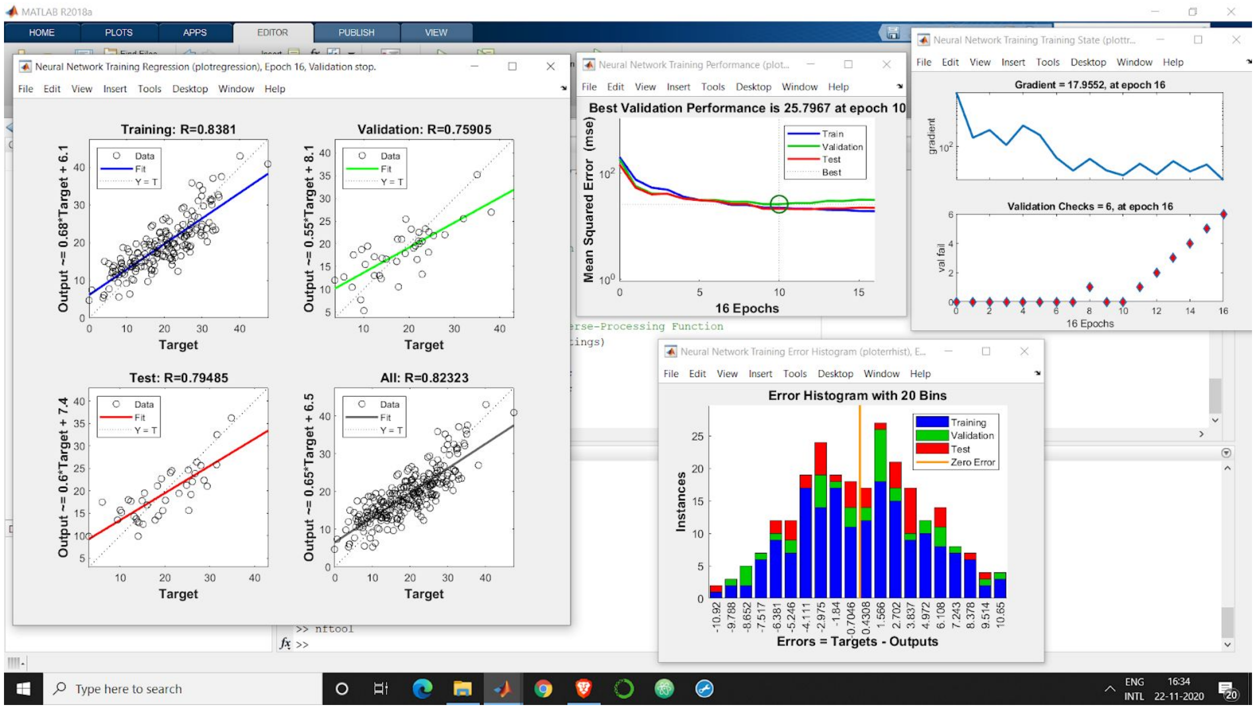
Matlab Algorithm's Code:

```
function [Y,Xf,Af] = myNeuralNetworkFunction(X,~,~)
%MYNEURALNETWORKFUNCTION neural network simulation function.
% Generated by Neural Network Toolbox function genFunction, 22-Nov-2020 16:30:22.
% [Y] = myNeuralNetworkFunction(X,~,~) takes these arguments:
%   X = 1xTS cell, 1 inputs over TS timesteps
%   Each X{1,ts} = 13xQ matrix, input #1 at timestep ts.
% and returns:
%   Y = 1xTS cell of 1 outputs over TS timesteps.
%   Each Y{1,ts} = 1xQ matrix, output #1 at timestep ts.
% where Q is number of samples (or series) and TS is the number of timesteps.

% ===== NEURAL NETWORK CONSTANTS =====

% Input 1
x1_step1.xoffset = [22;118.5;29.5;31.1;79.3;69.4;85;47.2;33;19.1;24.8;21;15.8];
x1_step1.gain = [0.0338983050847458;0.00817494379726139;0.0414507772020725;0.099502487562189;0.0351493848857645;0.0254129606099111;0.0318979266347687;0.0498753117206983;0.124223602484472;0.135135135135135;0.099009900990099;0.143884892086331;0.357142857142857];
x1_step1.ymin = -1;

% Layer 1
b1 = [-1.8005399867819646964;1.3513457495207454873;0.90520746138459973196;-0.56366084857895448756;-0.33021810621987973677;0.0069533531944103027511;0.58797401253544645527;-0.95706111006704086552;-1.3528988334078664302;-1.64095875812628611];
IW1_1 = [0.32251867238793358039 -0.27697350872347625828 0.28643822686243713971 0.2554145376139709267 -0.22534975790980937838 -0.37205766695188746684 -0.52331756720165401031 0.51368434452934808032 0.33974965707190973863
-0.54113842560077418664          0.62441863811404163531          0.45416614887470702078
-0.63629671396199871669;-0.39167106675355140011                  0.33430042575430196639
0.046582919710254820644          0.12460984555141885188          -0.20888938740226861701
0.69698505976225888503           -0.38238368978500336667         -0.41448794581803194426
-0.017529647557709904931         -0.80268928505356917924         0.73475838080465150082
0.58648602802688210023                 -0.17765007060894832946;-0.47677667756834407609
0.66525020943891366443          -0.031454708229534472652         0.31296378307614436398
-0.76041841244408925338          -0.22776943470324681473          -0.87691866505882598857
0.050480559774477122559          0.26934962172285681348          0.13553352459940476438
0.046742240261287086589                                          0.66652208286154457362
0.29083777406015437483;0.27841642390895426917                    -0.32453397137933731598
0.64289764572067720216           -0.19059116767577904961          -0.44312437886161493315
-0.81020880209313650422          -0.46805606874418459462          0.69886302775813358146
-0.23706297956097033275          -0.11379073344053659833          0.28289459684633344594
-0.036228645046263781293                0.71512301998115579416;0.36148874044859019561
0.48899247075524737705           -0.42782250088848411407          0.23599999480961914022
-0.43111436929528529349          -0.22364286651228881819          0.47784520494390470002
0.45073640964045802448           -0.73000537183981395462          0.75873222280342533796
0.15688089805215960082                                          0.067622711011789751745
0.57855718015453694303;0.34961137681762483043                    -0.52075751559744509755
-0.29537195134465266122          0.49526676111052647666           -0.11733912713788798021
0.59635236734106367162           0.43617004760491823179           0.61902656358503271861
-0.087454159784967760993         0.61580428381383855996           -0.68508771009828772769
0.37248028286956075261                 -0.088437571981364121454;0.23516019221600997779
0.53447087260058279146           0.6548689114214290008            -0.69979580586486411775
-0.36765799037994734144          0.59837844633402292871            0.60426399829784205322
0.0017819484770462607794         -0.21205555577812248 98          -0.53510791905408583435
0.033207362898577293 12                                          0.49936789163107186962
0.35688750490162352014;-0.74359847835788739356                   -0.59671385840340773754
-0.80064453140602265258          -0.70073370854876348979          -0.24083504646565034868
0.10078934581500190171           0.15560183759658630098          -0.032510348163901685303
0.49081747833702071837           0.49180561563383151658           0.25716498436347567935
0.25985267570994430297                 -0.14681911470299305322;-0.17398474674144395746
0.13652821473555087906           0.62177941249681645264           0.67181435101195563497
-0.52039040732013408519          0.29243069709171187753           -0.59356038847442815776
0.095722283558117071678          0.39180671951560902544           0.52880608142324680987
0.031278244464186803764                                          -0.47257037677655788777
0.65934207882060691386;-0.10314745059602827004                   -0.19749916497993991182
-0.49342171993066546998          0.28164036275695125688           0.67172932864109513584
0.68127890000678947846           0.73614685932496237708           -0.35914993083048490918
0.31067541614146387818           0.18230718161963027635           -0.18674494303276834017
0.29141561213819666687 0.85399011300969485116];

% Layer 2
b2 = 0.69143716118480202937;
LW2_1  =  [0.68478336593460809034  -0.15696731292936105806  -0.34490402666256891884
-0.56579420464933838364          0.38855874767680798065           0.40112820301509111154
0.68153192532445372454           0.11252125881036302568           0.10456797832973499518
-0.68579088321143455431];

% Output 1
y1_step1.ymin = -1;
```

```
y1_step1.gain = 0.0421052631578947;
y1_step1.xoffset = 0;

% ===== SIMULATION ========
% Format Input Arguments
isCellX = iscell(X);
if ~isCellX
   X = {X};
end
% Dimensions
TS = size(X,2); % timesteps
if ~isempty(X)
   Q = size(X{1},2); % samples/series
else
   Q = 0;
end
% Allocate Outputs
Y = cell(1,TS);
% Time loop
for ts=1:TS

   % Input 1
   Xp1 = mapminmax_apply(X{1,ts},x1_step1);

   % Layer 1
   a1 = tansig_apply(repmat(b1,1,Q) + IW1_1*Xp1);

   % Layer 2
   a2 = repmat(b2,1,Q) + LW2_1*a1;

   % Output 1
   Y{1,ts} = mapminmax_reverse(a2,y1_step1);
end
% Final Delay States
Xf = cell(1,0);
Af = cell(2,0);
% Format Output Arguments
if ~isCellX
   Y = cell2mat(Y);
end
end

% ===== MODULE FUNCTIONS ========
% Map Minimum and Maximum Input Processing Function
function y = mapminmax_apply(x,settings)
y = bsxfun(@minus,x,settings.xoffset);
y = bsxfun(@times,y,settings.gain);
y = bsxfun(@plus,y,settings.ymin);
end
% Sigmoid Symmetric Transfer Function
function a = tansig_apply(n,~)
a = 2 ./ (1 + exp(-2*n)) - 1;
```

```
end
% Map Minimum and Maximum Output Reverse-Processing Function
function x = mapminmax_reverse(y,settings)
x = bsxfun(@minus,y,settings.ymin);
x = bsxfun(@rdivide,x,settings.gain);
x = bsxfun(@plus,x,settings.xoffset);
end
```

Output Plots:


MATLAB R2018a
HOME
PLOTS
APPS
EDITOR
PUBLISH
VIEW
Neural Network Training Regression (plotregression), Epoch 16, Validation stop.
Training: R=0.8381
Output ~= 0.68*Target + 6.1
Validation: R=0.75905
Output ~= 0.55*Target + 8.1
Test: R=0.79485
Output ~= 0.6*Target + 7.4
All: R=0.82323
Output ~= 0.65*Target + 6.5
Data
Fit
Y = T
Target
Neural Network Training Performance (plot...
Best Validation Performance is 25.7967 at epoch 10
Mean Squared Error (mse)
Train
Validation
Test
Best
16 Epochs
Neural Network Training Training State (plottr...
Gradient = 17.9552, at epoch 16
gradient
Validation Checks = 6, at epoch 16
val fail
16 Epochs
Neural Network Training Error Histogram (ploterrhist), E...
Error Histogram with 20 Bins
Instances
Training
Validation
Test
Zero Error
Errors = Targets - Outputs
>> nftool
Type here to search
ENG INTL 16:34 22-11-2020

Practical 4: Learning of different Fuzzy logic membership functions

Theory: When you build a fuzzy inference system, as described in Fuzzy Inference Process, you can replace the built-in membership functions, inference functions, or both with custom functions. In this section, we try to build a fuzzy inference system using custom functions in the Fuzzy Logic Designer app.

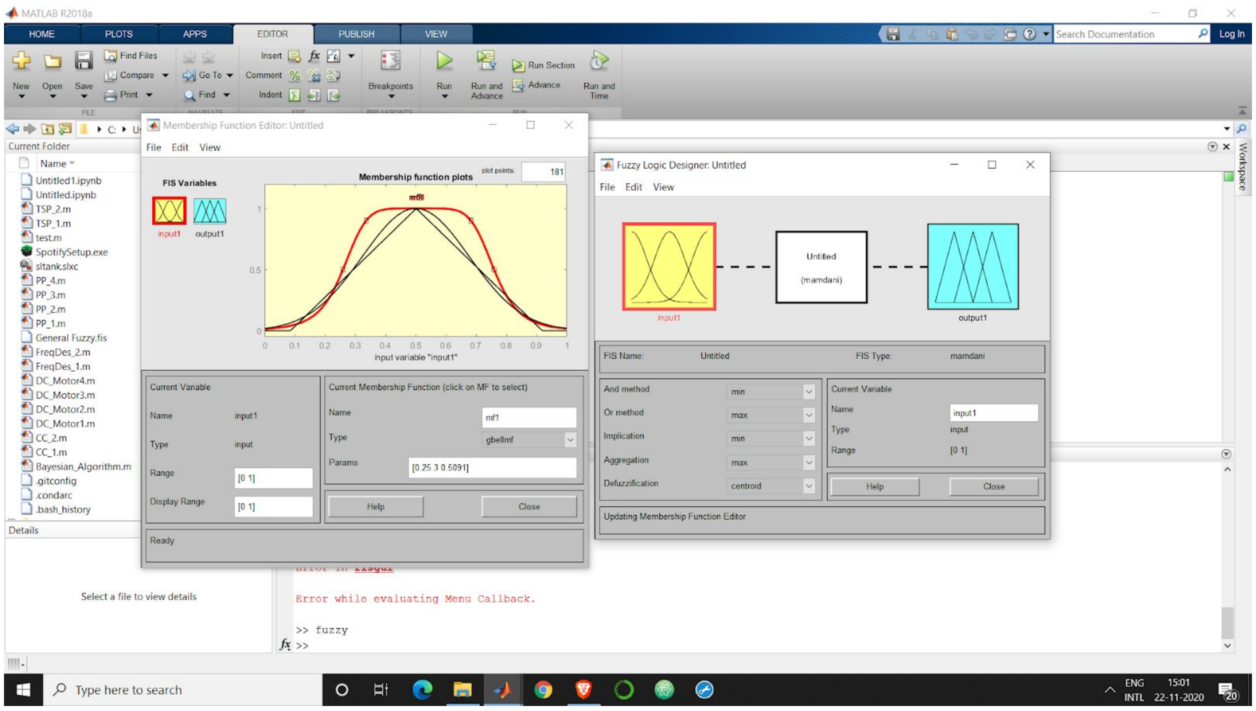
To build a fuzzy inference system using custom functions in the Fuzzy Logic Designer Toolbox:

1. Open Fuzzy Logic Designer. At the MATLAB command line.

2. Specify the number of inputs and outputs of the fuzzy system, as described in Fuzzy Logic Designer.

3. Create custom membership functions, and replace the built-in membership functions with them. Membership functions define how each point in the input space is mapped to a membership value between 0 and 1.

   Input membership functions:



Membership Function Editor: Untitled

Current Variable — Name: input1; Type: input; Range: [0 1]; Display Range: [0 1]

Current Membership Function (click on MF to select) — Name: mf1; Type: gbellmf; Params: [0.25 3 0.5091]

Ready

Fuzzy Logic Designer: Untitled

FIS Name: Untitled FIS Type: mamdani

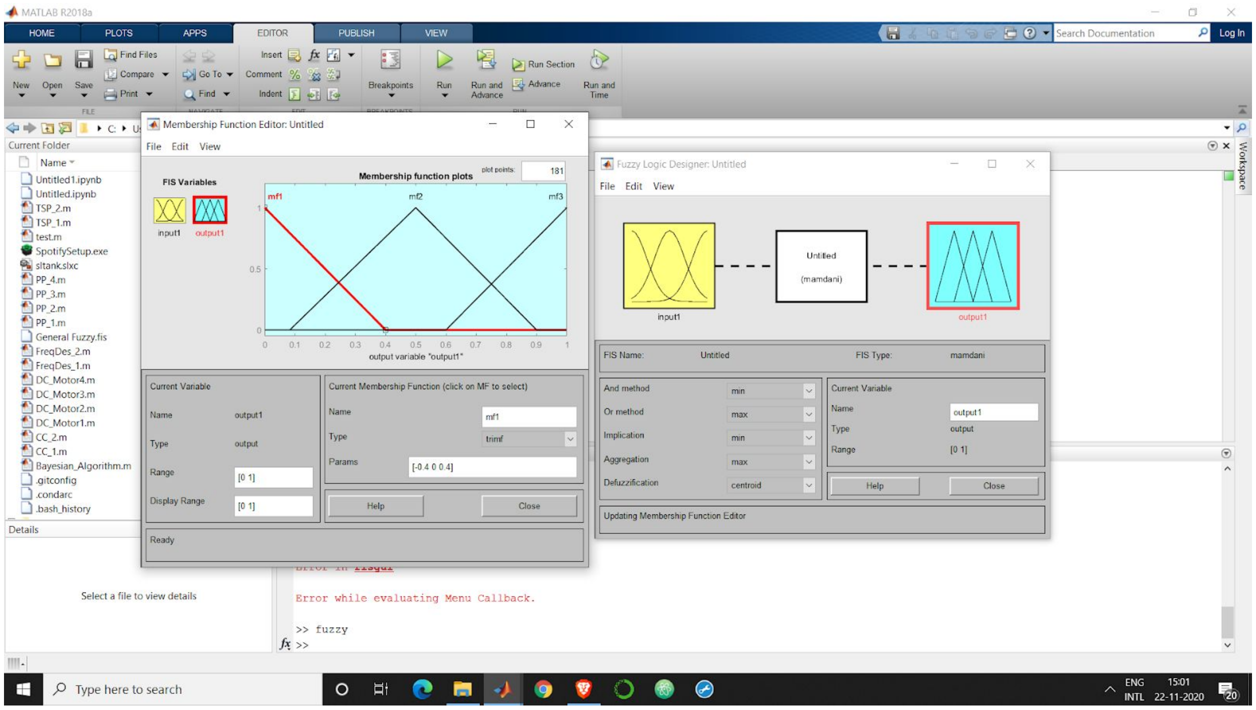
And method	min
Or method	max
Implication	min
Aggregation	max
Defuzzification	centroid

Current Variable — Name: input1; Type: input; Range: [0 1]

Updating Membership Function Editor

```
Error while evaluating Menu Callback.

>> fuzzy
>>
```

   Output membership functions:


Membership Function Editor: Untitled
File Edit View
FIS Variables
input1
output1
Membership function plots
plot points: 181
mf1
mf2
mf3
output variable "output1"
Fuzzy Logic Designer: Untitled
Untitled
(mamdani)
Help
Close

Current Variable — Name: output1; Type: output; Range: [0 1]; Display Range: [0 1]

Current Membership Function (click on MF to select) — Name: mf1; Type: trimf; Params: [-0.4 0 0.4]

Ready

FIS Name: Untitled FIS Type: mamdani

And method: min; Or method: max; Implication: min; Aggregation: max; Defuzzification: centroid

Current Variable — Name: output1; Type: output; Range: [0 1]

Updating Membership Function Editor

```
Error while evaluating Menu Callback.

>> fuzzy
>>
```

4. Create rules using the Rule Editor.
   Rules define the logical relationship between the inputs and the outputs.
5. Create custom inference functions, and replace the built-in inference functions with them.
   Inference methods include the AND, OR, implication, aggregation, and defuzzification methods.

The description of the Fuzzy System is:

Name='Untitled'
Type='mamdani'
Version=2.0
NumInputs=1
NumOutputs=1
NumRules=3
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'

Practical 5: To implement a Fuzzy logic rule base for a Washing Machine

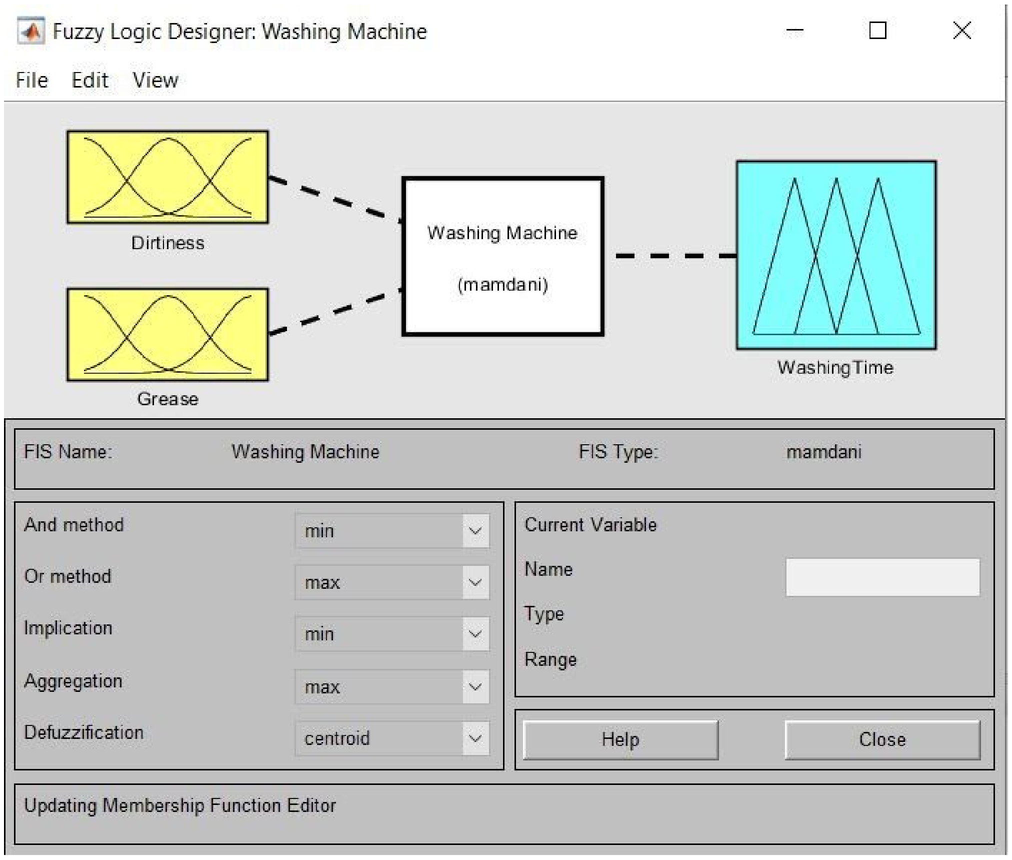
Theory:

The goal is to create a fuzzy logic controller for a washing machine, that will compute the appropriate washing time, based on certain information about the clothes that are to be washed.

The inputs of the system are: **The degree of dirtiness** - can be determined by examining the transparency of the water.

Grease - can be determined by examining the soaking time needed for the water to get to constant transparency - transparency saturation.

The structure of the Fuzzy Logic System is:



FIS Name: Washing Machine

FIS Type: mamdani

And method: min

Or method: max

Implication: min

Aggregation: max

Defuzzification: centroid

Updating Membership Function Editor

Input and Output Variables:

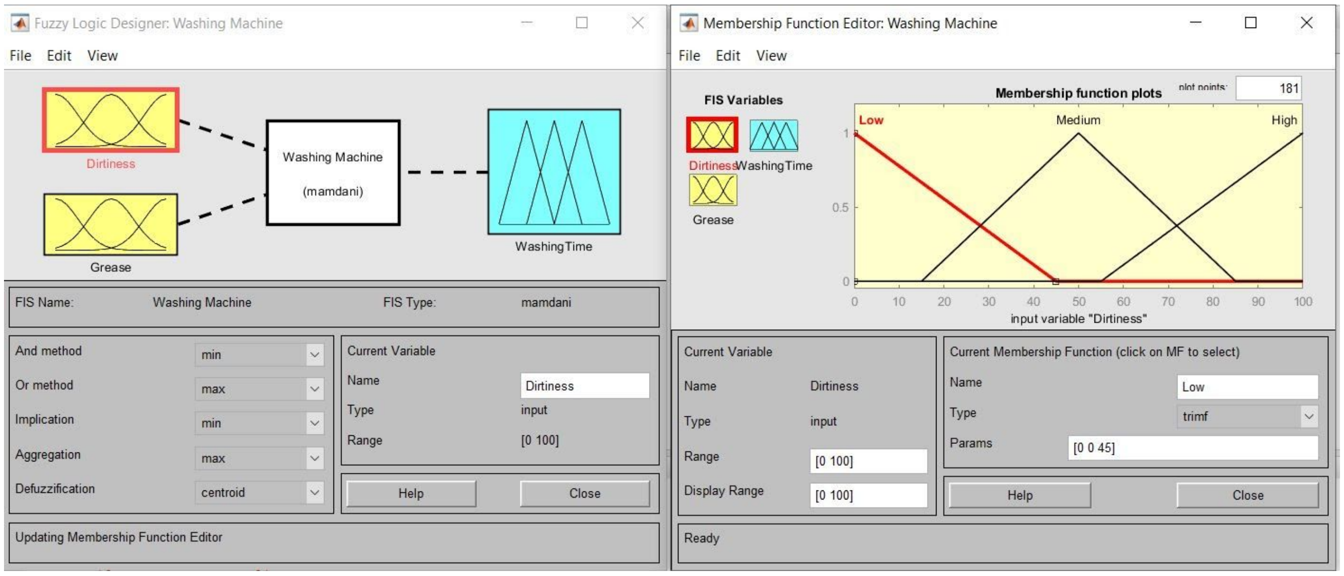
The first input is:

Variable Name: "Dirtiness"

Range=[0 100]

Num of MFs=3

MF1='Low':'trimf',[0 0 45]
MF2='Medium':'trimf',[15 50 85]



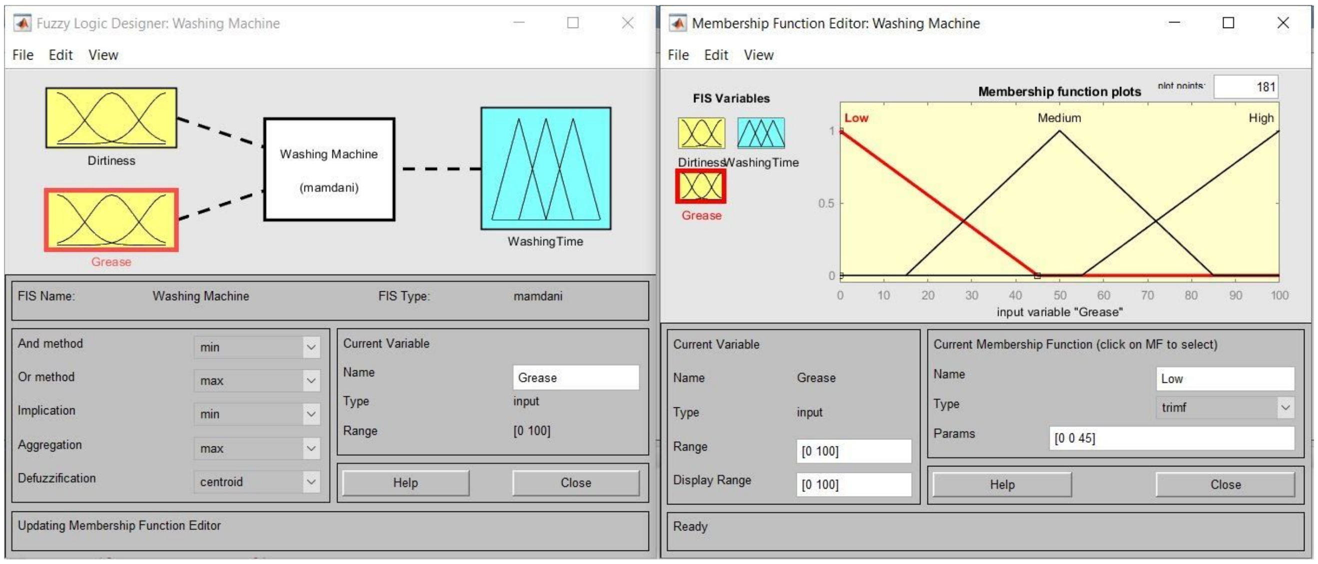
FIS Name: Washing Machine
FIS Type: mamdani
And method: min
Or method: max
Implication: min
Aggregation: max
Defuzzification: centroid
Current Variable
Name: Dirtiness
Type: input
Range: [0 100]
Updating Membership Function Editor

Current Variable
Name: Dirtiness
Type: input
Range: [0 100]
Display Range: [0 100]
Current Membership Function (click on MF to select)
Name: Low
Type: trimf
Params: [0 0 45]
Ready

The second input is:

Variable Name: “Grease”

Range=[0 100]
Num of MFs=3
MF1='Low':'trimf',[0 0 45]
MF2='Medium':'trimf',[15 50 85]
MF3='High':'trimf',[55 100 100]



FIS Name: Washing Machine
FIS Type: mamdani
And method: min
Or method: max
Implication: min
Aggregation: max
Defuzzification: centroid
Current Variable
Name: Grease
Type: input
Range: [0 100]
Updating Membership Function Editor

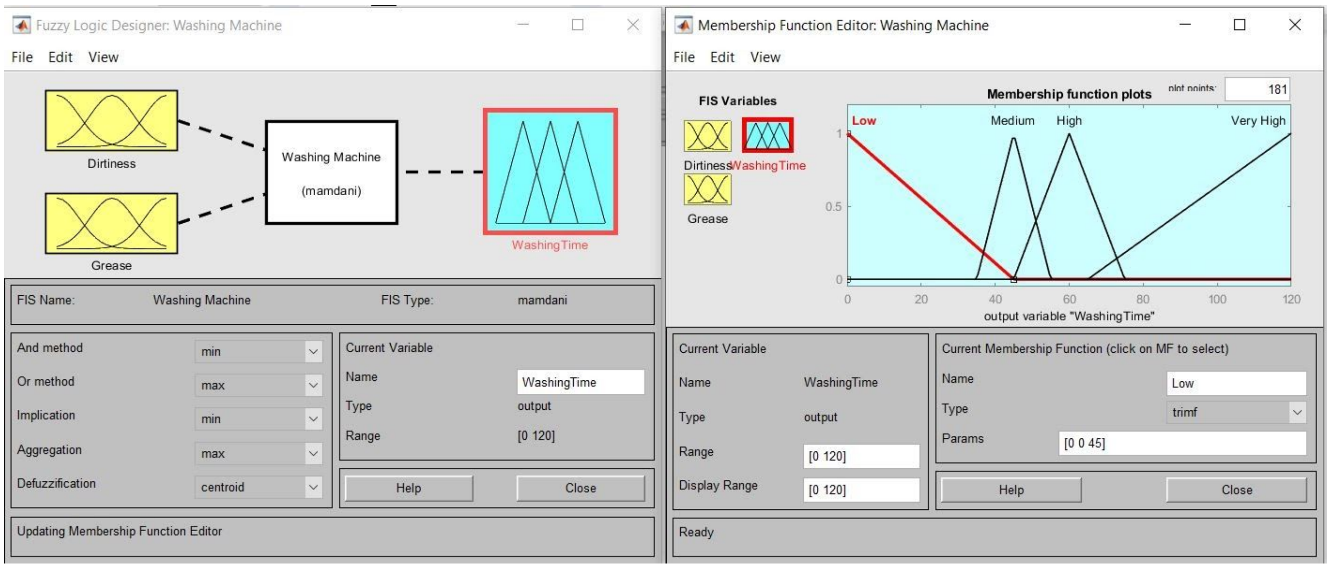
Current Variable
Name: Grease
Type: input
Range: [0 100]
Display Range: [0 100]
Current Membership Function (click on MF to select)
Name: Low
Type: trimf
Params: [0 0 45]
Ready

The output is:

Variable Name: “WashingTime”

Range=[0 120]
Num of MFs=4
MF1='Low':'trimf',[0 0 45]
MF2='Medium':'trimf',[35 45 55]
MF3='High':'trimf',[45 60 75]

MF4='VeryHigh':'trimf',[65 120 120]



FIS Name: Washing Machine

FIS Type: mamdani

And method: min

Or method: max

Implication: min

Aggregation: max

Defuzzification: centroid

Current Variable

Name: WashingTime

Type: output

Range: [0 120]

Updating Membership Function Editor

Current Variable

Name: WashingTime

Type: output

Range: [0 120]

Display Range: [0 120]

Current Membership Function (click on MF to select)

Name: Low

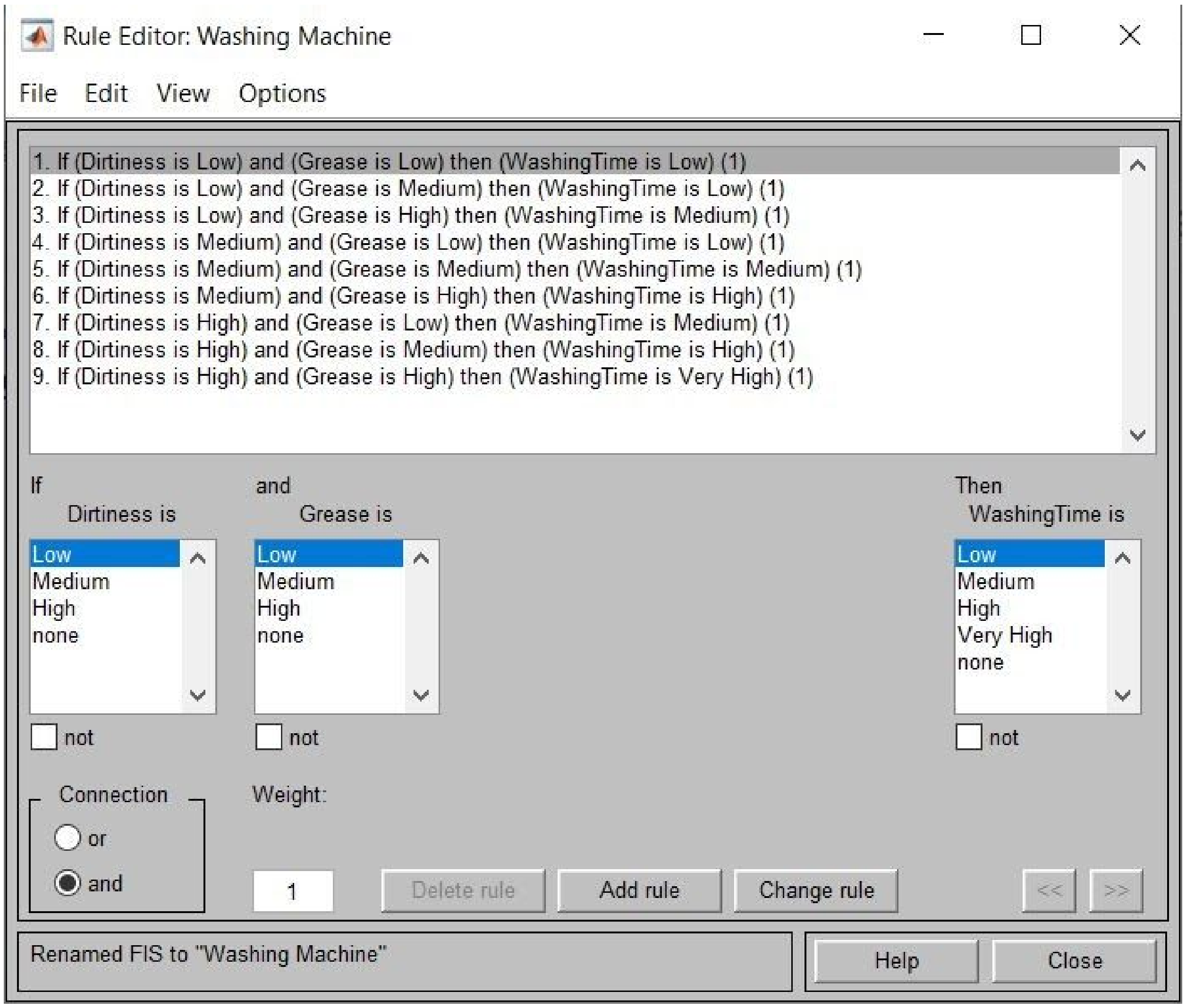
Type: trimf

Params: [0 0 45]

Ready

Rules for the Washing Machine

The fuzzy rules for this application are:



1. If (Dirtiness is Low) and (Grease is Low) then (WashingTime is Low) (1)
2. If (Dirtiness is Low) and (Grease is Medium) then (WashingTime is Low) (1)
3. If (Dirtiness is Low) and (Grease is High) then (WashingTime is Medium) (1)
4. If (Dirtiness is Medium) and (Grease is Low) then (WashingTime is Low) (1)
5. If (Dirtiness is Medium) and (Grease is Medium) then (WashingTime is Medium) (1)
6. If (Dirtiness is Medium) and (Grease is High) then (WashingTime is High) (1)
7. If (Dirtiness is High) and (Grease is Low) then (WashingTime is Medium) (1)
8. If (Dirtiness is High) and (Grease is Medium) then (WashingTime is High) (1)
9. If (Dirtiness is High) and (Grease is High) then (WashingTime is Very High) (1)

Connection: or [✓ and]

Weight: 1

Renamed FIS to "Washing Machine"

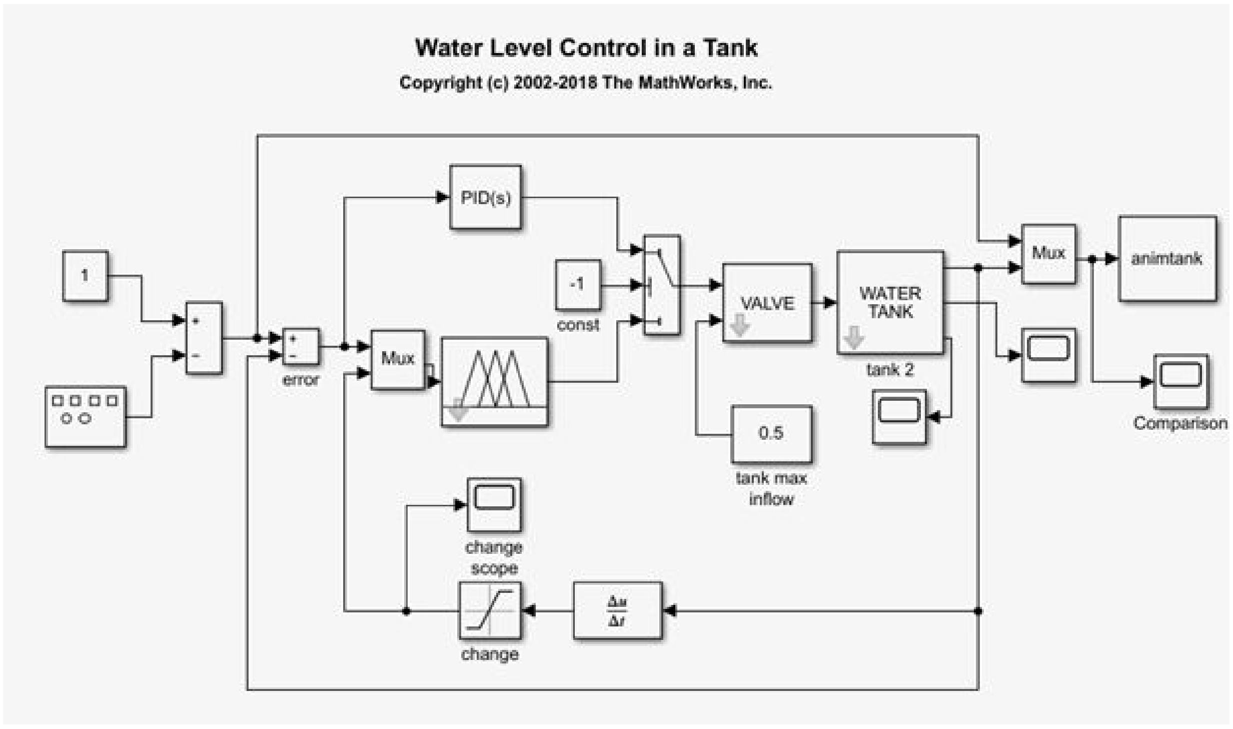
Practical 6: To implement a Fuzzy logic rule base for a Water Tank

Theory:

This experiment presents a detailed description of simulation of water level control in a tank using fuzzy logic, which clears that fuzzy logic is a different way to represent linguistic and subjective attributes of the real world.

In order to improve the efficiency and simplicity of the design process, fuzzy logic can be applied to simulation of water level control in a tank using MATLAB.

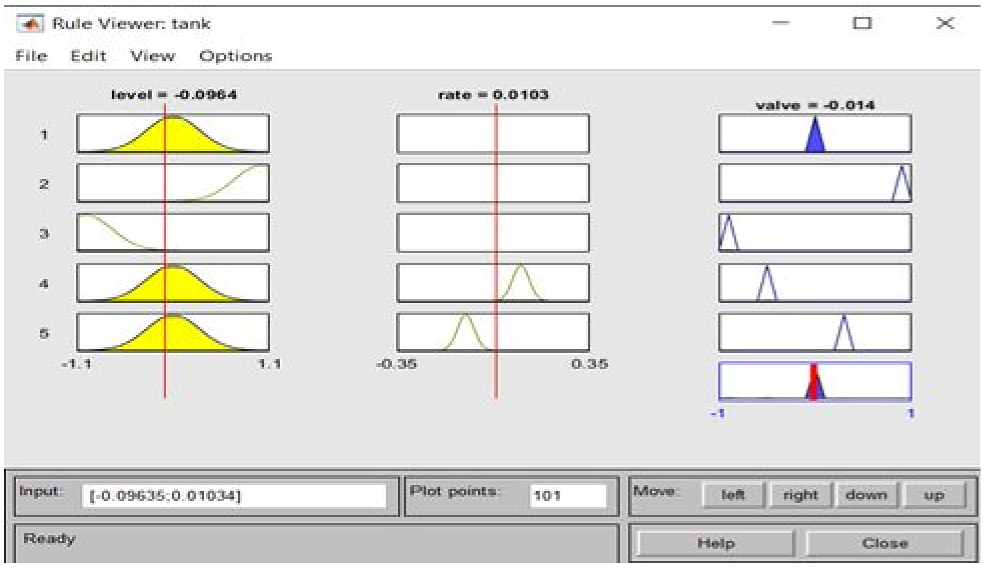
This paper design a simulation system of fuzzy logic controller for water tank level control by using Fuzzy Logic Toolbox MATLAB. This proves that fuzzy logic does a fairly better job than other controlling systems.



The fuzzy system has five rules. The first three rules adjust the valve based on only the water level error.

- If the water level is okay, then do not adjust the valve.
- If the water level is low, then open the valve quickly.
- If the water level is high, then close the valve quickly.
- The other two rules adjust the valve based on the rate of change of the water level when the water level is near the setpoint.
- If the water level is okay and increasing, then close the valve slowly.
- If the water level is okay and decreasing, then open the valve slowly.

RuleViewer:

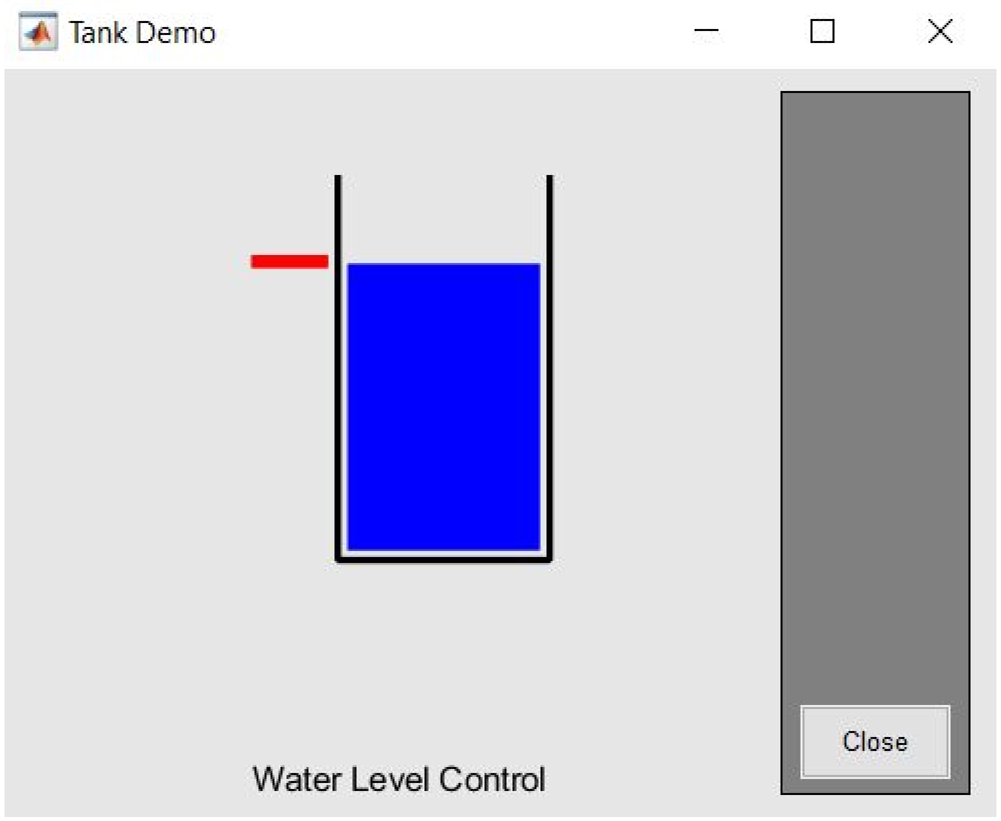


Input: [-0.09635;0.01034]

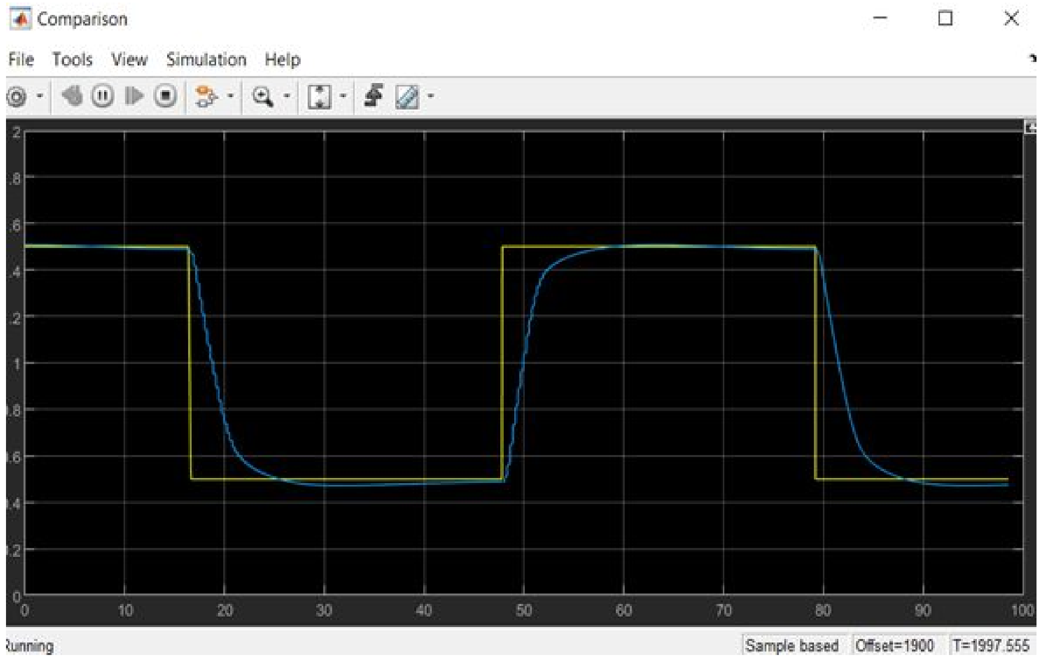
Plot points: 101

Ready

Tank Demo:

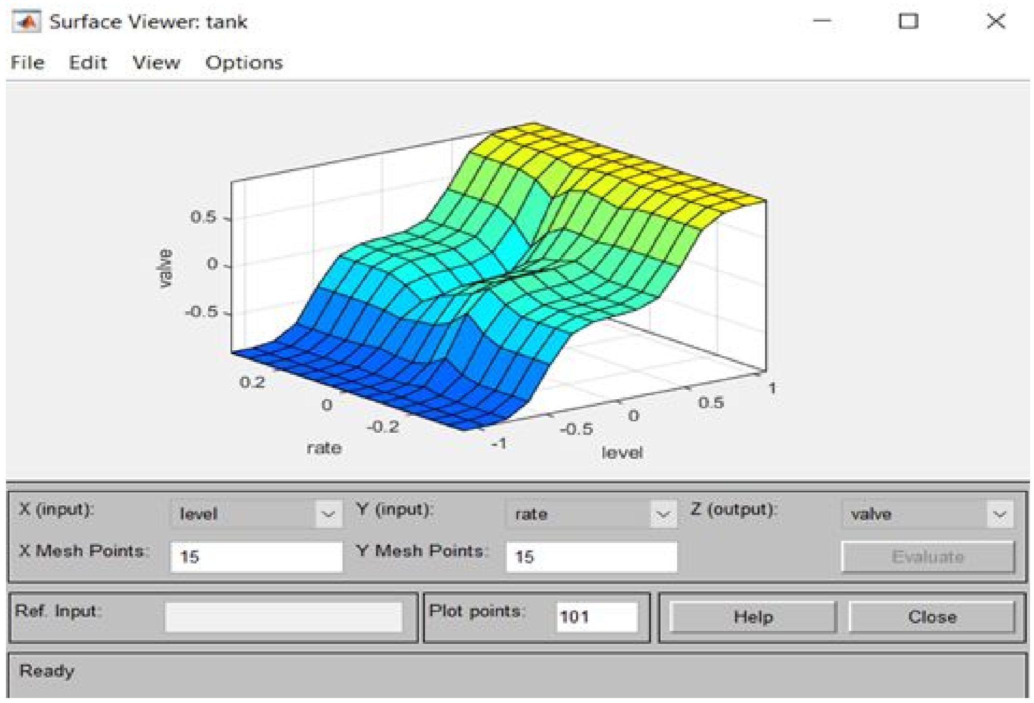


Output Waveform:



Blue line: Output **Yellow line**: Input

View:



X (input): level
Y (input): rate
Z (output): valve
X Mesh Points: 15
Y Mesh Points: 15
Ref. Input:
Plot points: 101

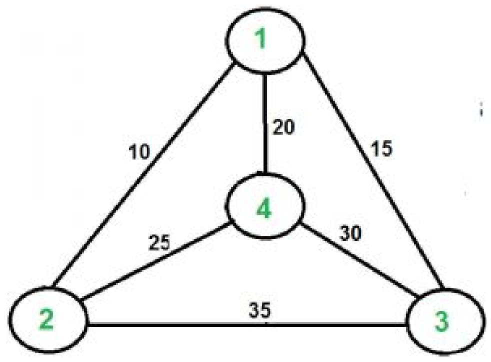
Practical 7: To implement the Traveling Salesman Problem using search algorithms.

Theory:

The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics.
Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

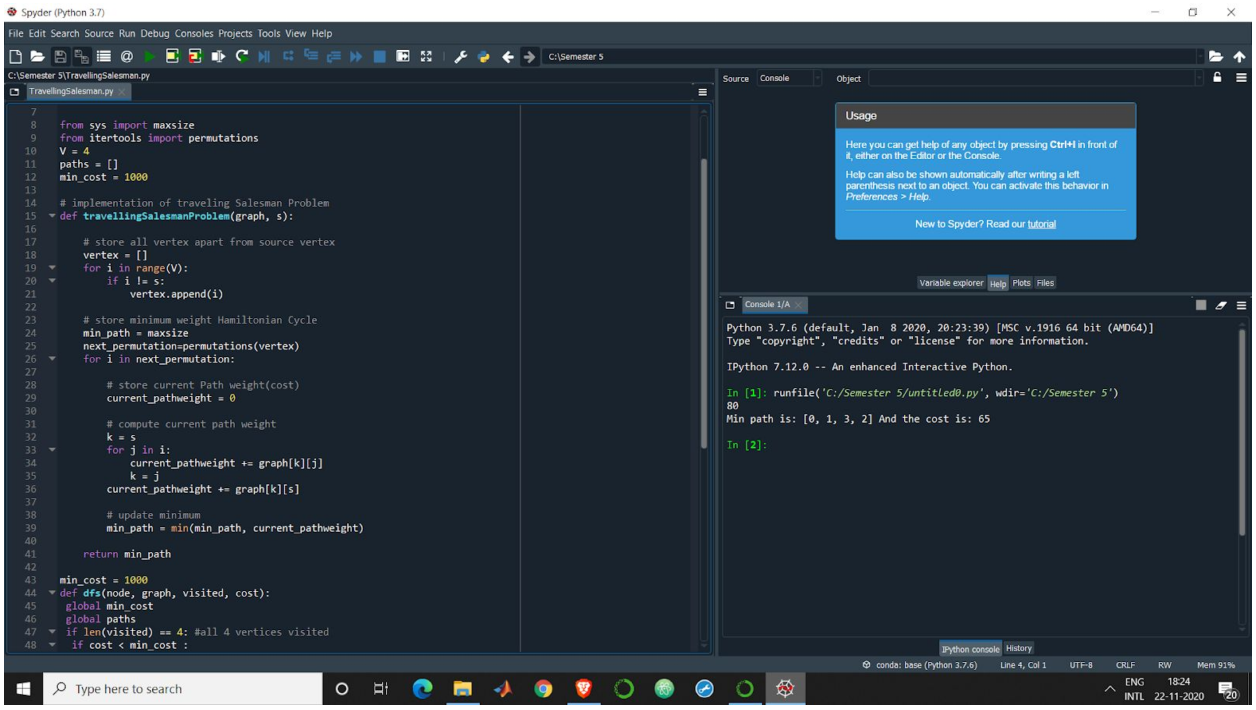
Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that the Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.
For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. **The cost of the tour is 10+25+30+15 which is 80.**

The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem.



Output of Given Graph:
minimum weight Hamiltonian Cycle :
10 + 25 + 30 + 15 := 80
In this post, the **implementation** of a simple solution is discussed.

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.



```python
from sys import maxsize
from itertools import permutations
V = 4
paths = []
min_cost = 1000

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path

min_cost = 1000
def dfs(node, graph, visited, cost):
 global min_cost
 global paths
 if len(visited) == 4: #all 4 vertices visited
  if cost < min_cost :
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.12.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Semester 5/untitled0.py', wdir='C:/Semester 5')
80
Min path is: [0, 1, 3, 2] And the cost is: 65

In [2]:
```

The Python 3.6 code for the same is:

```python
from sys import maxsize
from itertools import permutations
V = 4
paths = []
min_cost = 1000

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path

min_cost = 1000
def dfs(node, graph, visited, cost):
 global min_cost
 global paths
 if len(visited) == 4: #all 4 vertices visited
  if cost < min_cost :
       #print(visited,":",cost)
       paths = list(visited)
       min_cost = cost
  return
 for i in range(4):
     if i not in visited:
        visited.append(i)
        cost = cost + graph[node][i]
        dfs(i, graph, visited, cost)
        cost = cost - graph[node][i]
        visited.pop(len(visited)-1)

# Driver Code
def main():
    # matrix representation of graph
    global paths
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
            [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
    visited = []
    dfs(0,graph, visited, 0)
    print("Min path is:",paths,"And the cost is:",min_cost)

main()
```

Practical 8: To create the Adaline Neural Network

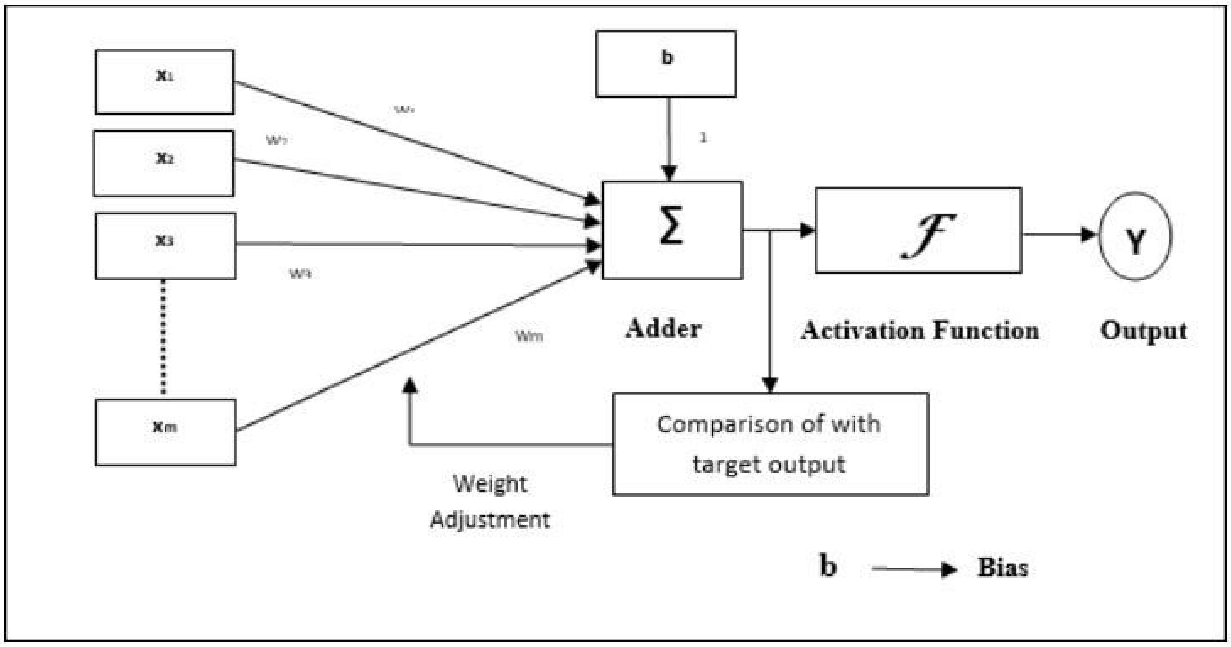
Theory:

Adaline which stands for Adaptive Linear Neuron, is a network having a single linear unit. It was developed by Widrow and Hoff in 1960. Some important points about Adaline are as follows −

- It uses bipolar activation function.
- It uses delta rule for training to minimize the Mean-Squared Error MSE
- MSE between the actual output and the desired/target output.
- The weights and the bias are adjustable.

Architecture:

The basic structure of Adaline is similar to perceptron having an extra feedback loop with the help of which the actual output is compared with the desired/target output. After comparison on the basis of the training algorithm, the weights and bias will be updated.

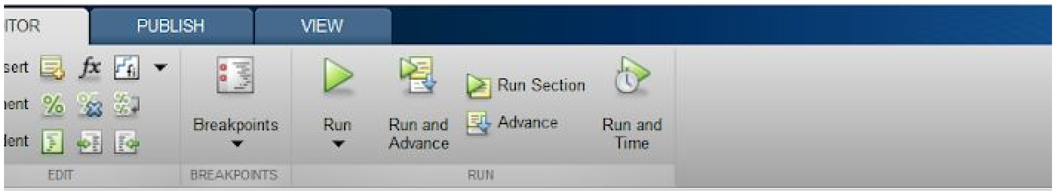


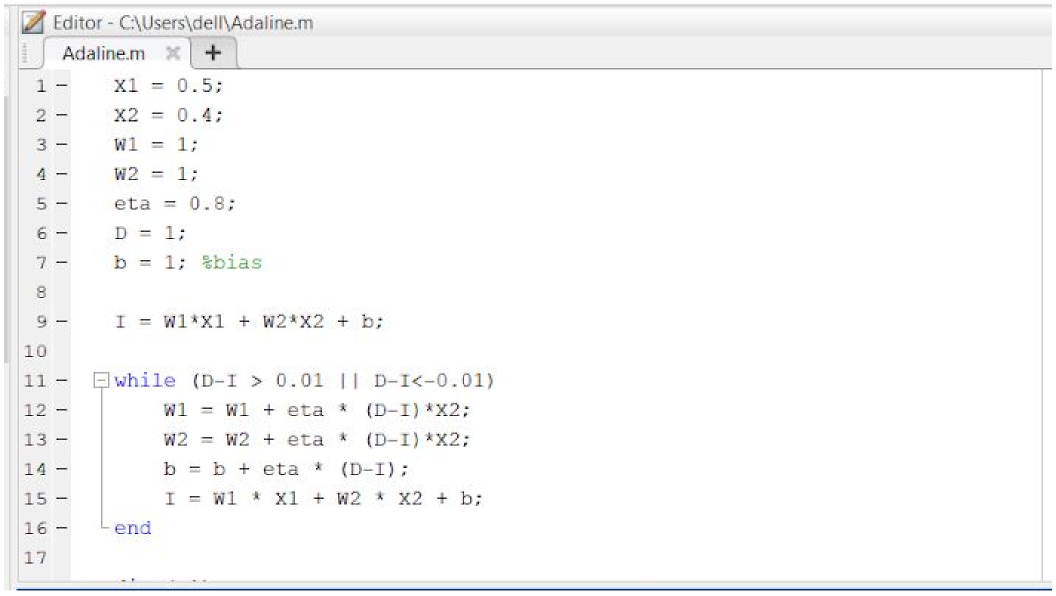
Adaline is a single-unit neuron, which receives input from several units and also from one unit, called bias. An Adeline model consists of trainable weights. The inputs are of two values (+1 or -1) and the weights have signs (positive or negative).

Initially random weights are assigned. The net input calculated is applied to a quantizer transfer function (possibly activation function) that restores the output to +1 or -1. The Adaline model compares the actual output with the target output and with the bias and adjusts all the weights.

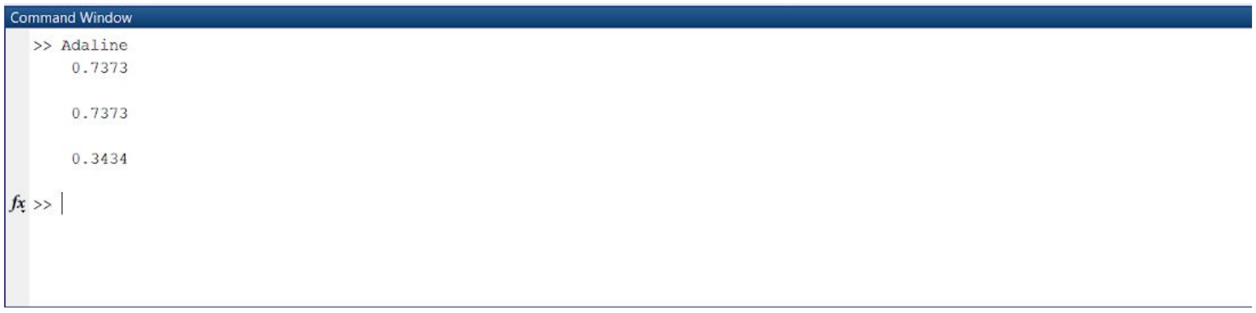
The difference between Adaline and the standard (McCulloch–Pitts) perceptron is that in the learning phase, the weights are adjusted according to the weighted sum of the inputs (the net). In the standard perceptron, the net is passed to the activation (transfer) function and the function's output is used for adjusting the weights.

A multilayer network of ADALINE units is known as a MADALINE.





```
X1 = 0.5;
X2 = 0.4;
W1 = 1;
W2 = 1;
eta = 0.8;
D = 1;
b = 1; %bias

I = W1*X1 + W2*X2 + b;

while (D-I > 0.01 || D-I<-0.01)
    W1 = W1 + eta * (D-I)*X2;
    W2 = W2 + eta * (D-I)*X2;
    b = b + eta * (D-I);
    I = W1 * X1 + W2 * X2 + b;
end

```



```
>> Adaline
     0.7373

     0.7373

     0.3434

>>
```

Fig: Screenshots of MATLAB code and output screen for Adaline

PRACTICAL 9: To implement Back Propagation Algorithm

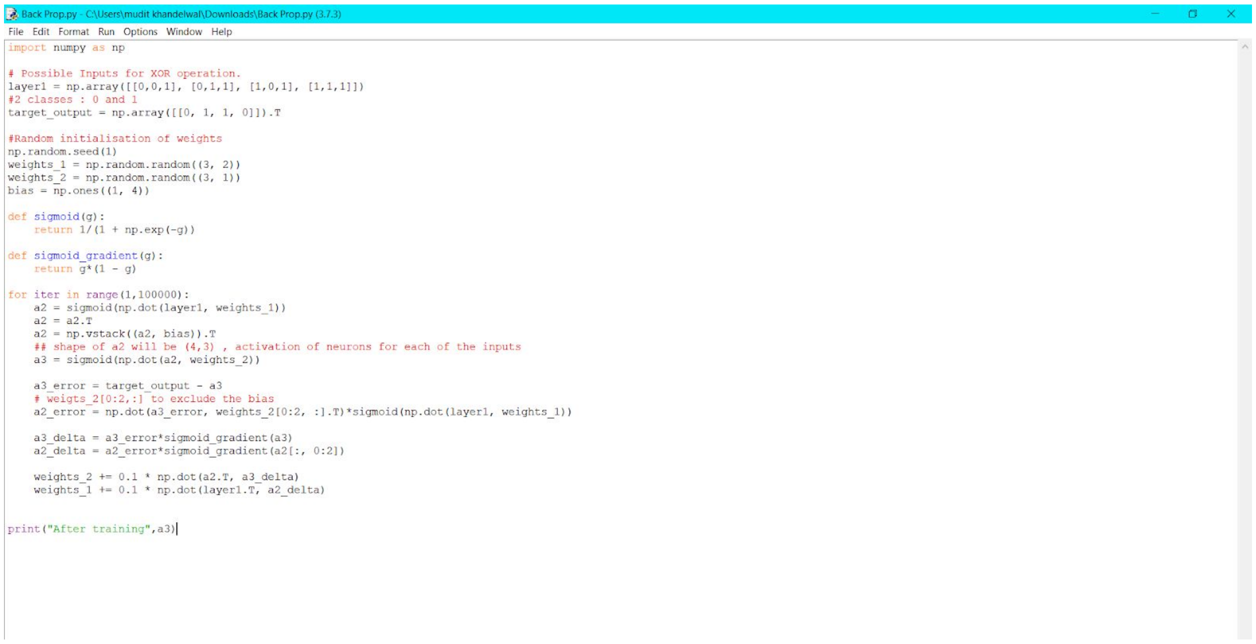
Theory:

Backpropagation is a supervised learning algorithm, for training Multi-layer Perceptrons (Artificial Neural Networks). The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem.

Procedure:

The following steps summarize the steps to implement the backpropagation algorithm:

- **Calculate the error** – How far is your model output from the actual output.
- **Minimum Error** – Check whether the error is minimized or not.
- **Update the parameters** – If the error is huge then, update the parameters (weights and biases). After that again check the error. Repeat the process until the error becomes minimum.
- **Model is ready to make a prediction** – Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.


Back Prop.py - C:\Users\mudit khandelwal\Downloads\Back Prop.py (3.7.3)
File Edit Format Run Options Window Help

```python
import numpy as np

# Possible Inputs for XOR operation.
layer1 = np.array([[0,0,1], [0,1,1], [1,0,1], [1,1,1]])
#2 classes : 0 and 1
target_output = np.array([[0, 1, 1, 0]]).T

#Random initialisation of weights
np.random.seed(1)
weights_1 = np.random.random((3, 2))
weights_2 = np.random.random((3, 1))
bias = np.ones((1, 4))

def sigmoid(g):
    return 1/(1 + np.exp(-g))

def sigmoid_gradient(g):
    return g*(1 - g)

for iter in range(1,100000):
    a2 = sigmoid(np.dot(layer1, weights_1))
    a2 = a2.T
    a2 = np.vstack((a2, bias)).T
    ## shape of a2 will be (4,3) , activation of neurons for each of the inputs
    a3 = sigmoid(np.dot(a2, weights_2))

    a3_error = target_output - a3
    # weigts_2[0:2,:] to exclude the bias
    a2_error = np.dot(a3_error, weights_2[0:2, :].T)*sigmoid(np.dot(layer1, weights_1))

    a3_delta = a3_error*sigmoid_gradient(a3)
    a2_delta = a2_error*sigmoid_gradient(a2[:, 0:2])

    weights_2 += 0.1 * np.dot(a2.T, a3_delta)
    weights_1 += 0.1 * np.dot(layer1.T, a2_delta)


print("After training",a3)
```

Fig: Screenshot of the python code for error back propagation

The Python 3.6 code for the same is:

Created on Sun Nov 15 11:51:55 2020

import numpy as np

Possible Inputs for XOR operation.

layer1 = np.array([[0,0,1], [0,1,1], [1,0,1], [1,1,1]])

```
#2 classes : 0 and 1

target_output = np.array([[0, 1, 1, 0]]).T


#Random initialisation of weights

np.random.seed(1)

weights_1 = np.random.random((3, 2))

weights_2 = np.random.random((3, 1))

bias = np.ones((1, 4))


def sigmoid(g):

    return 1/(1 + np.exp(-g))


def sigmoid_gradient(g):

    return g*(1 - g)



for iter in range(1,100000 ):

    a2 = sigmoid(np.dot(layer1, weights_1))

    a2 = a2.T

    a2 = np.vstack((a2, bias)).T

    ## shape of a2 will be (4,3) , activation of neurons for each of the inputs

    a3 = sigmoid(np.dot(a2, weights_2))

    a3_error = target_output - a3

    # weigts_2[0:2,:] to exclude the bias

    a2_error = np.dot(a3_error, weights_2[0:2, :].T)*sigmoid(np.dot(layer1, weights_1))
    a3_delta = a3_error*sigmoid_gradient(a3)

    a2_delta = a2_error*sigmoid_gradient(a2[:, 0:2])



    weights_2 += 0.1 * np.dot(a2.T, a3_delta)

    weights_1 += 0.1 * np.dot(layer1.T, a2_delta)

print("After training",a3)
```

CODE ENDS

Practical 10: To demonstrate Constrained Minimization using the Genetic Algorithm

Theory:

Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to the next generation. In simple words, they simulate "survival of the fittest" among individuals of consecutive generations for solving a problem. Each generation consists of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Problem:

We want to minimize a simple fitness function of two variables x1 and x2

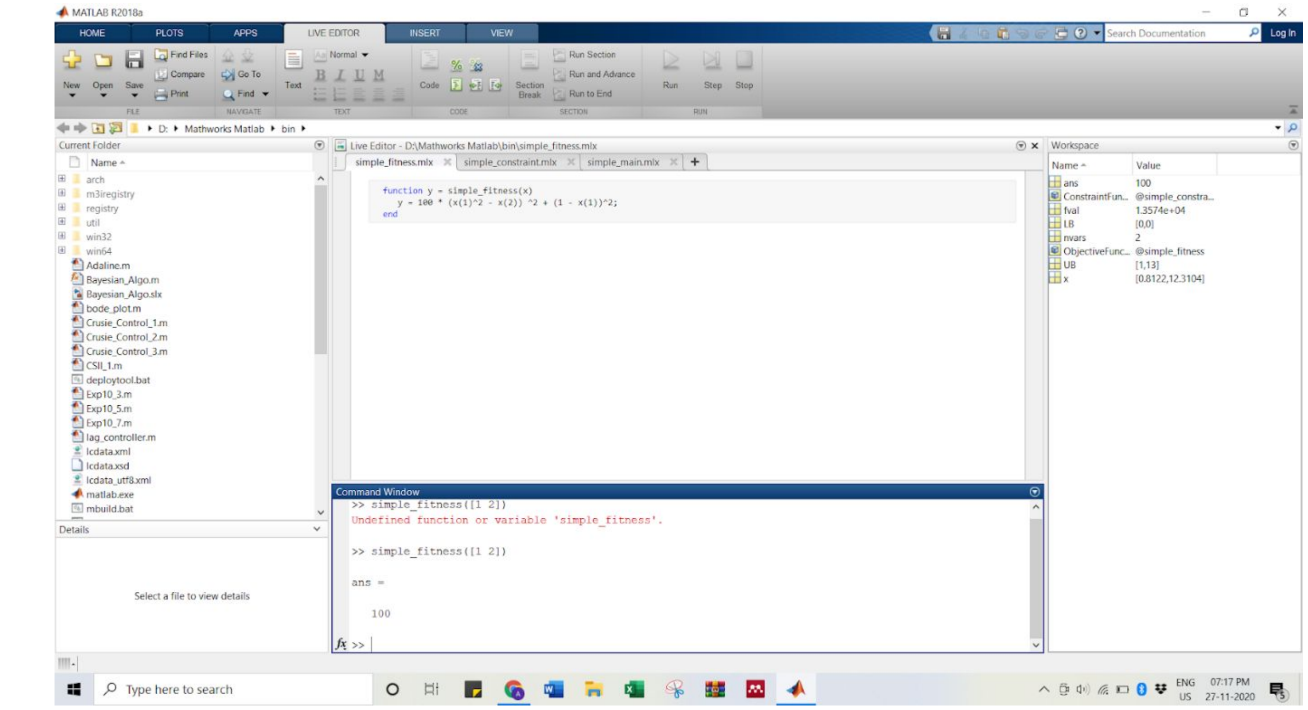
min f(x) = 100 * (x1^2 - x2) ^2 + (1 - x1)^2;

x such that the following two nonlinear constraints and bounds are satisfied

```
 x1*x2 + x1 - x2 + 1.5 <=0,  (nonlinear constraint)

  10 - x1*x2 <=0,            (nonlinear constraint)

  0 <= x1 <= 1, and          (bound)

  0 <= x2 <= 13              (bound)
```

The above fitness function is known as 'cam' as described in L.C.W. Dixon and G.P. Szego (eds.), Towards Global Optimisation 2, North-Holland, Amsterdam, 1978.

Output Plots:



Live Editor - D:\Mathworks Matlab\bin\simple_fitness.mlx

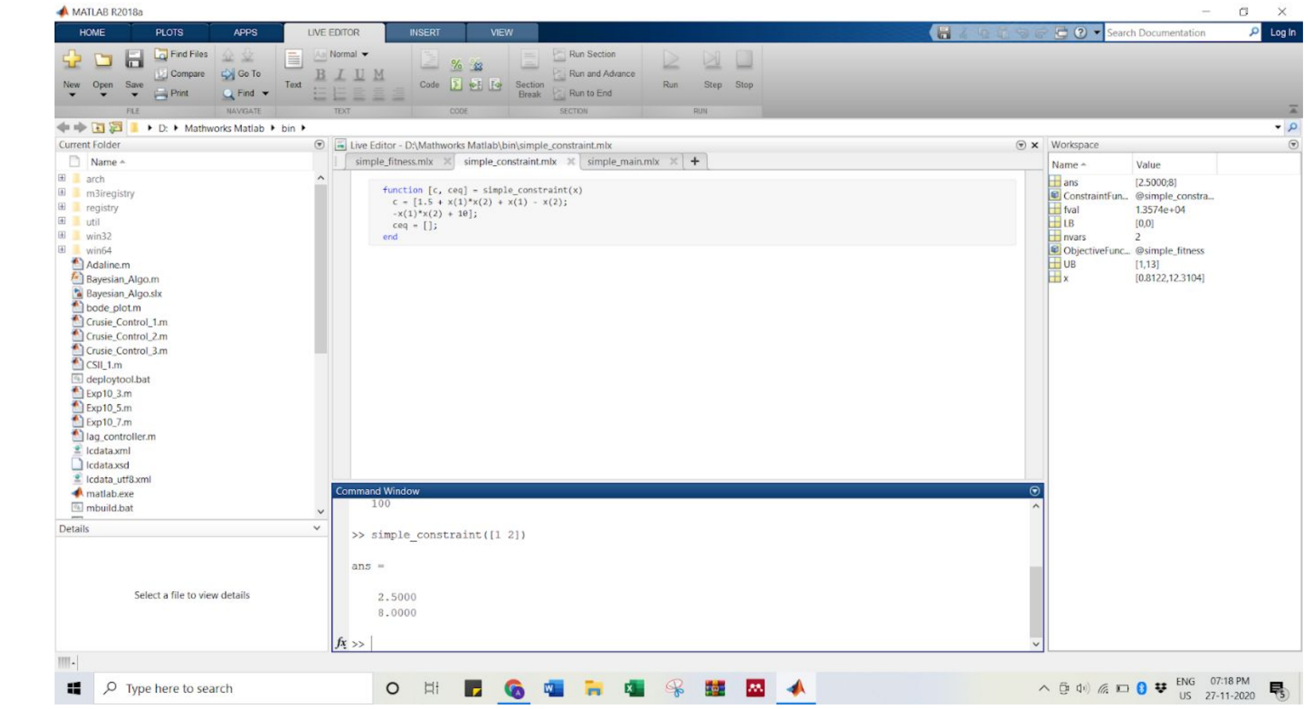
```
function y = simple_fitness(x)
   y = 100 * (x(1)^2 - x(2)) ^2 + (1 - x(1))^2;
end
```

Command Window

```
>> simple_fitness([1 2])
Undefined function or variable 'simple_fitness'.

>> simple_fitness([1 2])

ans =

   100

fx >>
```

Workspace

Name	Value
ans	100
ConstraintFun...	@simple_constra...
fval	1.3574e+04
LB	[0,0]
nvars	2
ObjectiveFunc...	@simple_fitness
UB	[1,13]
x	[0.8122,12.3104]



Live Editor - D:\Mathworks Matlab\bin\simple_constraint.mlx

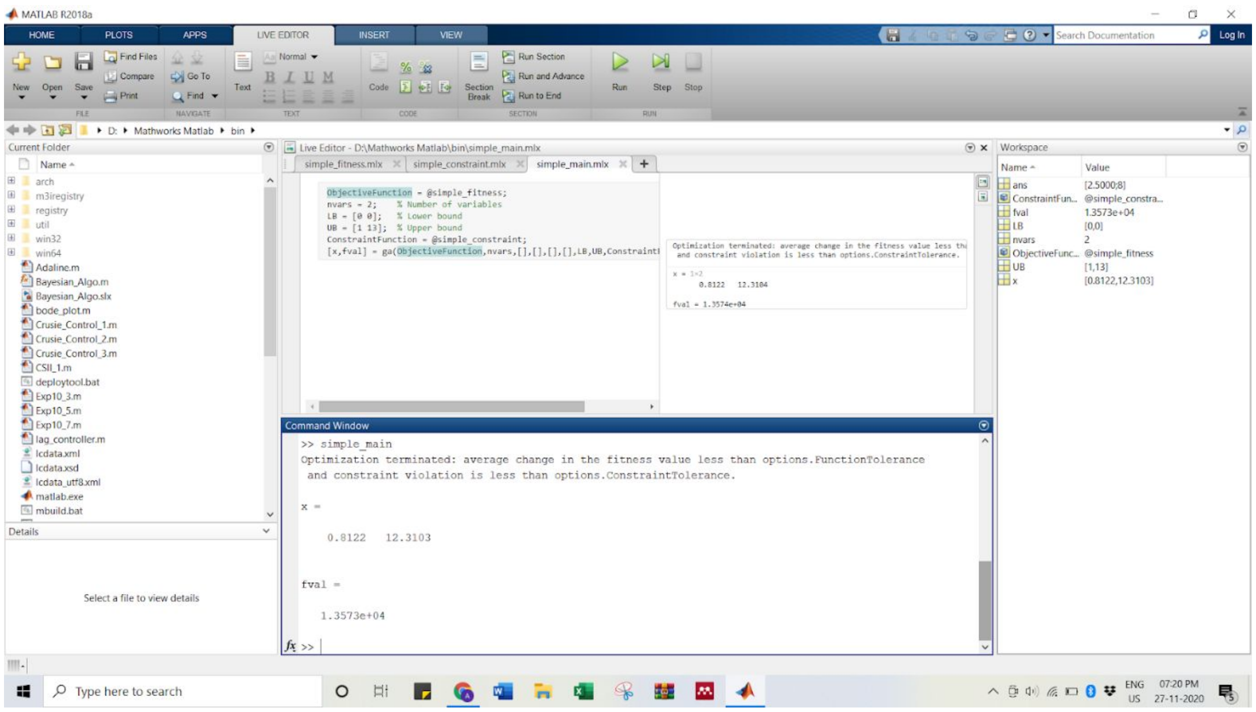
```
function [c, ceq] = simple_constraint(x)
  c = [1.5 + x(1)*x(2) + x(1) - x(2);
  -x(1)*x(2) + 10];
  ceq = [];
end
```

Command Window

```
   100

>> simple_constraint([1 2])

ans =

    2.5000
    8.0000

fx >>
```

Workspace

Name	Value
ans	[2.5000;8]
ConstraintFun...	@simple_constra...
fval	1.3574e+04
LB	[0,0]
nvars	2
ObjectiveFunc...	@simple_fitness
UB	[1,13]
x	[0.8122,12.3104]



```
ObjectiveFunction = @simple_fitness;
nvars = 2;    % Number of variables
LB = [0 0];   % Lower bound
UB = [1 13];  % Upper bound
ConstraintFunction = @simple_constraint;
[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB,Constraint
```

```
Optimization terminated: average change in the fitness value less th
 and constraint violation is less than options.ConstraintTolerance.

x = 1×2
      0.8122   12.3104

fval = 1.3574e+04
```

```
>> simple_main
Optimization terminated: average change in the fitness value less than options.FunctionTolerance
 and constraint violation is less than options.ConstraintTolerance.

x =

    0.8122   12.3103


fval =

   1.3573e+04

fx >>
```

Name	Value
ans	[2.5000;8]
ConstraintFun...	@simple_constra...
fval	1.3573e+04
LB	[0,0]
nvars	2
ObjectiveFunc...	@simple_fitness
UB	[1,13]
x	[0.8122,12.3103]

Matlab Algorithm's Code:

```
 function y = simple_fitness(x)

    y = 100 * (x(1)^2 - x(2)) ^2 + (1 - x(1))^2;

 end

function [c, ceq] = simple_constraint(x)

   c = [1.5 + x(1)*x(2) + x(1) - x(2);

   -x(1)*x(2) + 10];

   ceq = [];

 end

ObjectiveFunction = @simple_fitness;

nvars = 2;    % Number of variables

LB = [0 0];   % Lower bound

UB = [1 13];  % Upper bound

ConstraintFunction = @simple_constraint;

[x,fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB,ConstraintFunction)
```