Intelligent Systems Lab

Michele Baldassini

Department of Information Engineering – University of Pisa

michele.baldassini@unifi.it

Simple Neuron

Calculate the output of a simple neuron

- Define neuron parameters

```
close all, clear all, clc, format compact
% Neuron weights
w = [4 -2]
% Neuron bias
b = -3
% Activation function
func = 'tansig'
% func = 'purelin'
% func = 'hardlim'
% func = 'logsig'
```

- Define input vector

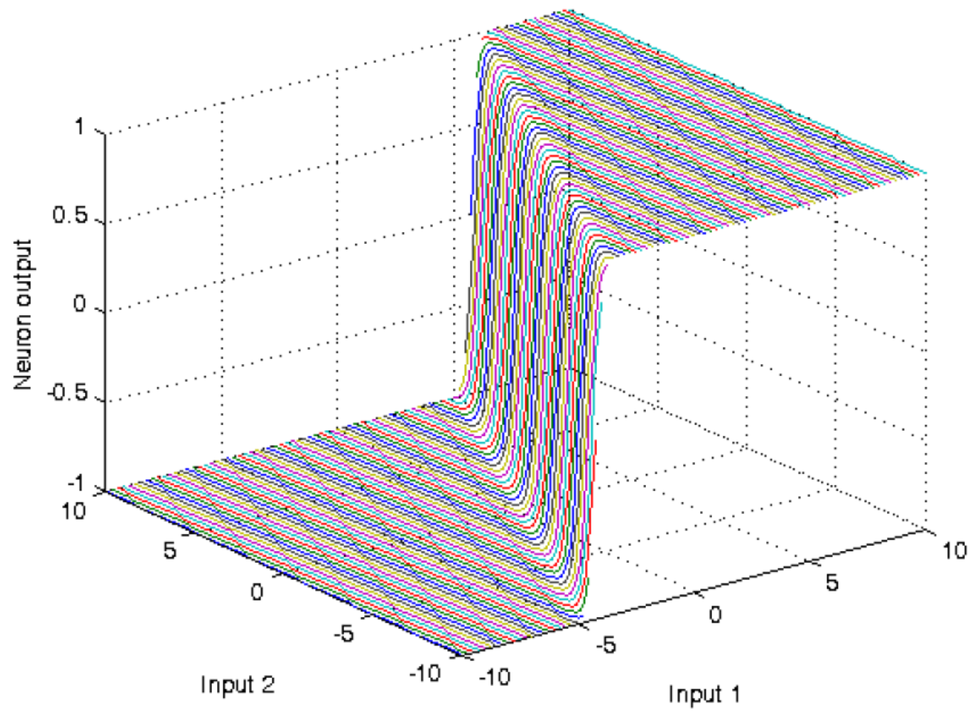
```
p = [2 3]
```

- Calculate neuron output

```
activation_potential = p*w'+b
neuron_output = feval(func, activation_potential)
```

- Plot neuron output over the range of inputs

```
[p1, p2] = meshgrid(-10:.25:10);
z = feval(func, [p1(:) p2(:)]*w'+b );
z = reshape(z, length(p1), length(p2));
plot3(p1, p2, z)
grid on
xlabel('Input 1')
ylabel('Input 2')
zlabel('Neuron output')
```



Custom Neural Networks

Create Custom Neural Networks

- Define one sample: inputs and outputs

```
close all, clear all, clc, format compact
inputs = [1:6]'         % input vector (6-dimensional pattern)
outputs = [1 2]'        % corresponding target output vector
```

- Define and custom network

```
% create network
net = network( ...
1, ...                  % numInputs, number of inputs,
2, ...                  % numLayers, number of layers
[1; 0], ...             % biasConnect, numLayers-by-1 Boolean vector,
[1; 0], ...             % inputConnect, numLayers-by-numInputs Boolean matrix,
[0 0; 1 0], ...         % layerConnect, numLayers-by-numLayers Boolean matrix
[0 1] ...               % outputConnect, 1-by-numLayers Boolean vector
);

% View network structure
view(net);
```

Train Custom Neural Networks

- Define topology and transfer function

```
% number of hidden layer neurons
net.layers{1}.size = 5;
% hidden layer transfer function
net.layers{1}.transferFcn = 'logsig';
view(net);
```

- Define topology and transfer function

```
net = configure(net,inputs,outputs);
view(net);
```

- Train net and calculate neuron output

```
% initial network response without training
initial_output = net(inputs)
% network training
net.trainFcn = 'trainlm';
net.performFcn = 'mse';
net = train(net,inputs,outputs);
% network response after training
final_output = net(inputs)
```

```
initial_output =
     0
     0
final_output =
    1.0000
    2.0000
```

Perceptron

Classification of linearly separable data with a perceptron

- Define input and output data

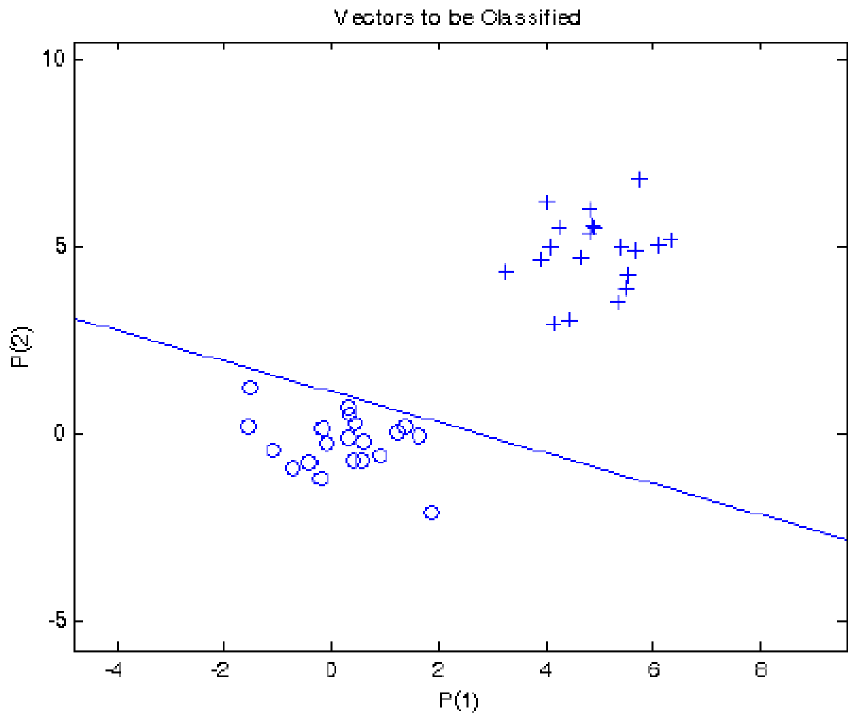
```
close all, clear all, clc, format compact
% number of samples of each class
N = 20;
% define inputs and outputs
offset = 5;                              % offset for second class
x = [randn(2,N) randn(2,N)+offset];      % inputs
y = [zeros(1,N) ones(1,N)];              % outputs
% Plot input samples with PLOTPV (Plot perceptron input/target vectors)
figure(1)
plotpv(x,y);
```

- Create and train perceptron

```
net = perceptron;
net = train(net,x,y);
view(net);
```

- Plot decision boundary

```
figure(1)
plotpc(net.IW{1},net.b{1});
```



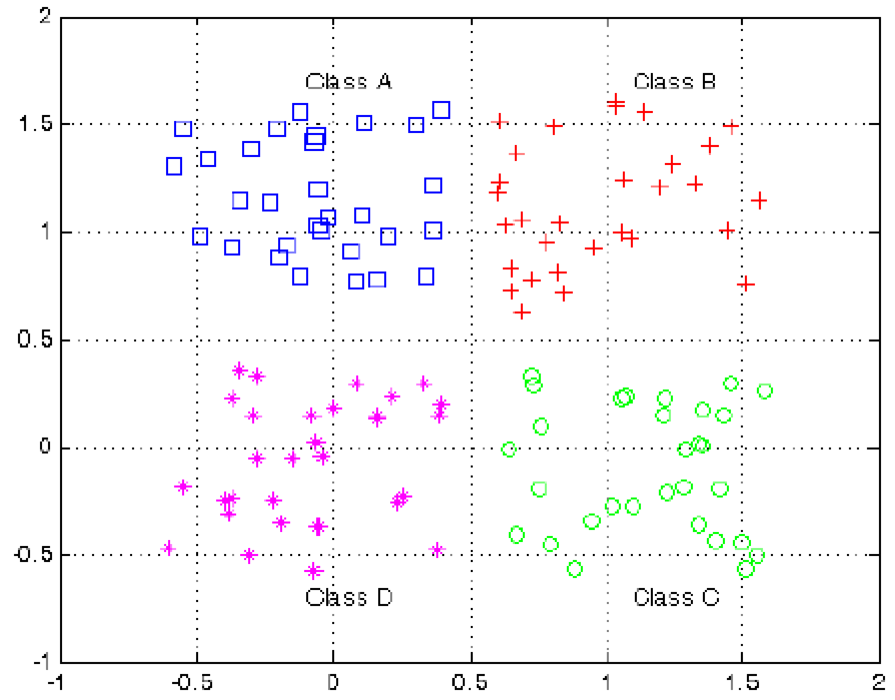
Classification of a 4-class problem with a perceptron (1/2)

- Define data

```
close all, clear all, clc, format compact
% number of samples of each class
K = 30;
% define classes
q = .6;            % offset of classes
A = [rand(1,K)-q; rand(1,K)+q];
B = [rand(1,K)+q; rand(1,K)+q];
C = [rand(1,K)+q; rand(1,K)-q];
D = [rand(1,K)-q; rand(1,K)-q];
% plot classes
plot(A(1,:),A(2,:),'bs')
hold on
grid on
plot(B(1,:),B(2,:),'r+')
plot(C(1,:),C(2,:),'go')
plot(D(1,:),D(2,:),'m*')
% text labels for classes
text(.5-q,.5+2*q,'Class A')
text(.5+q,.5+2*q,'Class B')
text(.5+q,.5-2*q,'Class C')
text(.5-q,.5-2*q,'Class D')
% define output coding for classes
a = [0 1]';
b = [1 1]';
c = [1 0]';
d = [0 0]';
```

- Prepare inputs & outputs for perceptron training

```
% define inputs (combine samples from all four classes)
P = [A B C D];
% define targets
T = [repmat(a,1,length(A)) repmat(b,1,length(B)) ...
      repmat(c,1,length(C)) repmat(d,1,length(D))];
%plotpv(P,T);
```



Classification of a 4-class problem with a perceptron (2/2)

- Create and train a perceptron

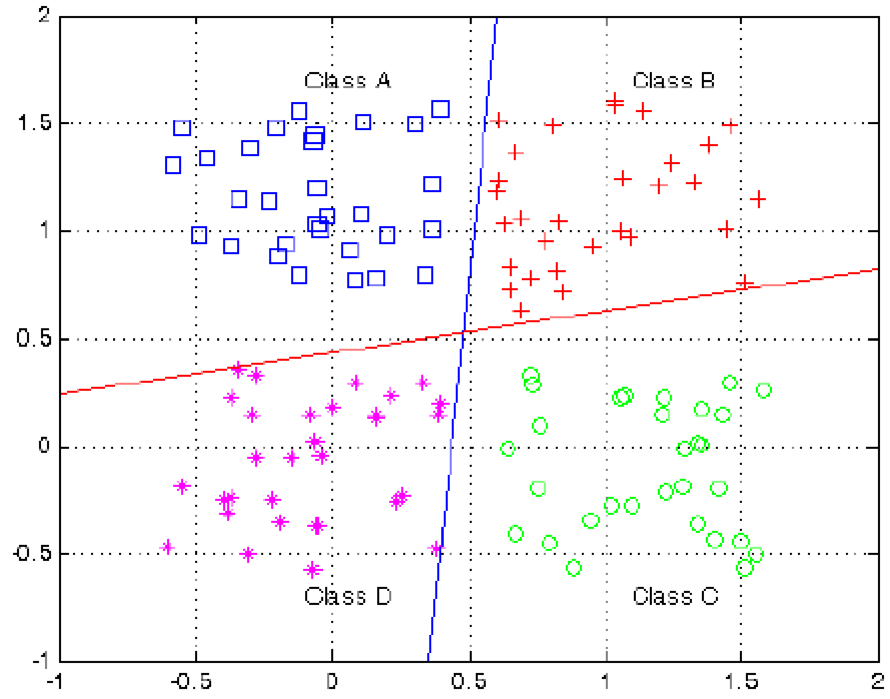
```
net = perceptron;
E = 1;
net.adaptParam.passes = 1;
linehandle = plotpc(net.IW{1},net.b{1});
n = 0;
while (sse(E) & n<1000)
    n = n+1;
    [net,Y,E] = adapt(net,P,T);
    linehandle = plotpc(net.IW{1},net.b{1},linehandle);
    drawnow;
end
% show perceptron structure
view(net);
```

ADAPT returns a new network object that performs as a better classifier, the network output, and the error.
This loop allows the network to adapt for xx passes,
plots the classification line and continues until the error is zero.

- How to use trained perceptron

```
% For example, classify an input vector of [0.7; 1.2]
p = [0.7; 1.2]
y = net(p)
% compare response with output coding (a,b,c,d)
```

```
p =
    0.7000
    1.2000
y =
    1
    1
```



Exercises Part 1

1. Train a perceptron to perform classification with the following inputs and outputs:
   ```
   X = [-0.5 -0.5 +0.3 -0.1; -0.5 +0.5 -0.5 +1.0];
   T = [1 1 0 0];
   ```
   Test with:
   ```
   x = [0.7; 1.2];
   ```

2. Train a perceptron to perform classification with the following inputs and outputs:
   ```
   X = [-0.5 -0.5 +0.3 -0.1 -40; -0.5 +0.5 -0.5 +1.0 50];
   T = [1 1 0 0 1];
   ```
   Test with:
   ```
   x = [0.7; 1.2];
   ```

Multilayer Perceptron

Generate feedforward neural network

net = feedforwardnet(hiddenSizes,trainFcn)

- returns a feedforward neural network with a hidden layer size of *hiddenSizes* and training function, specified by *trainFcn*
- Feedforward networks consist of a series of layers. The first layer has a connection from the network input. Each subsequent layer has a connection from the previous layer. The final layer produces the network's output.
- You can use feedforward networks for any kind of input to output mapping. A feedforward network with one hidden layer and enough neurons in the hidden layers can fit any finite input-output mapping problem.
- Specialized versions of the feedforward network include fitting and pattern recognition networks
- A variation on the feedforward network is the cascade forward network, which has additional connections from the input to every layer, and from each layer to all following layers

Train and Test

- Train the network net using the training data.

```
net = train(net, x, t);
```

- View the trained network.

```
view(net)
```

- Estimate the targets using the trained network.

```
y = net(x);
```

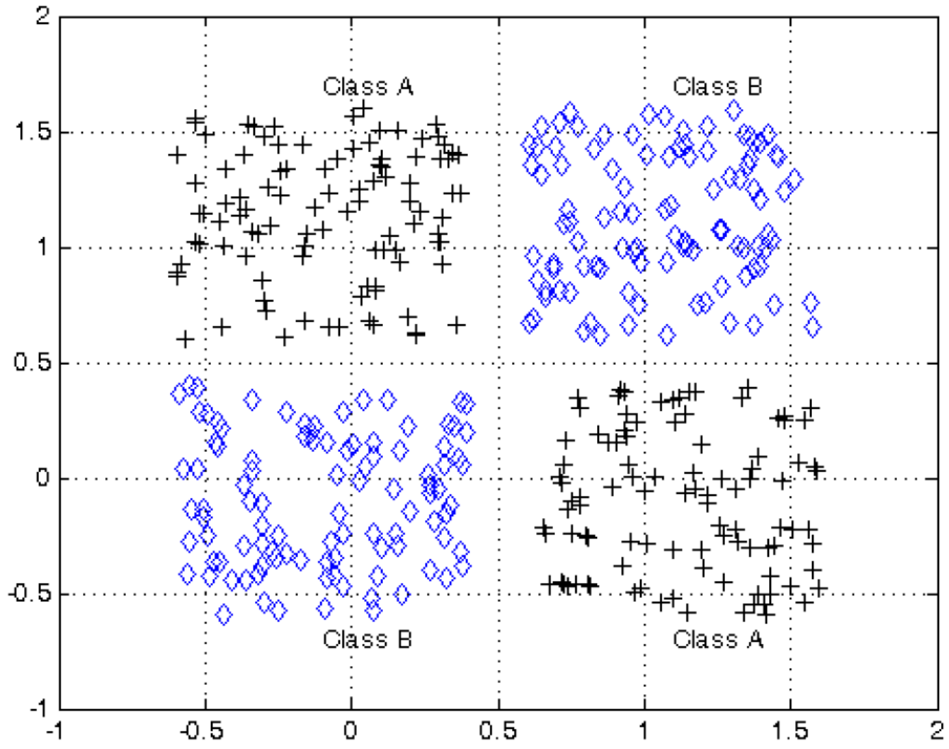
- Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net, y, t);
```

Solving XOR problem with a multilayer perceptron (1/4)

- Define 4 clusters of input data

```
close all, clear all, clc, format compact
% number of samples of each class
K = 100;
% define classes
q = .6;       % offset of classes
A = [rand(1,K)-q; rand(1,K)+q];
B = [rand(1,K)+q; rand(1,K)+q];
C = [rand(1,K)+q; rand(1,K)-q];
D = [rand(1,K)-q; rand(1,K)-q];
% plot classes
plot(A(1,:),A(2,:), 'k+')
hold on
grid on
plot(B(1,:),B(2,:), 'bd')
plot(C(1,:),C(2,:), 'k+')
plot(D(1,:),D(2,:), 'bd')
% text labels for classes
text(.5-q,.5+2*q,'Class A')
text(.5+q,.5+2*q,'Class B')
text(.5+q,.5-2*q,'Class A')
text(.5-q,.5-2*q,'Class B')
```



Solving XOR problem with a multilayer perceptron (2/4)

- Define output coding for XOR problem

```
% encode clusters a and c as one class, and b and d as another class
a = -1;          % a | b
c = -1;          % -------
b = 1;           % d | c
d = 1;           %
```

- Prepare inputs & outputs for network training

```
% define inputs (combine samples from all four classes)
P = [A B C D];
% define targets
T = [repmat(a,1,length(A)) repmat(b,1,length(B)) ...
     repmat(c,1,length(C)) repmat(d,1,length(D)) ];
% view inputs |outputs
%[P' T ']
```

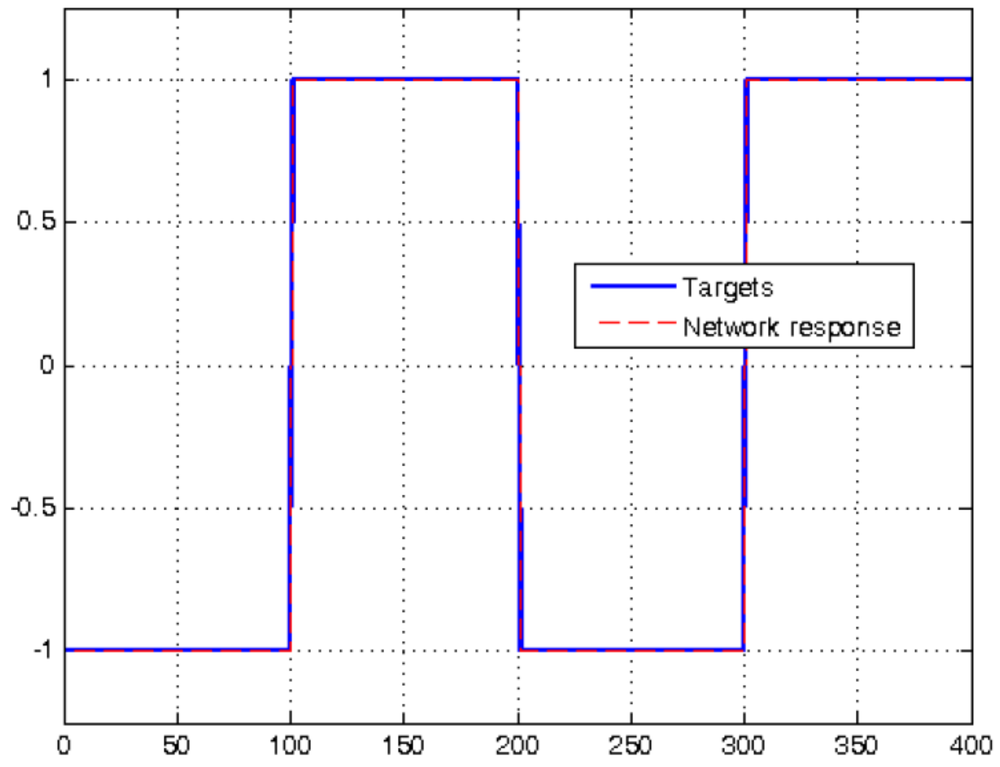
- Create and train a multilayer perceptron

```
% create a neural network
net = feedforwardnet([5 3]);
% train net
net.divideParam.trainRatio = 1;         % training set [%]
net.divideParam.valRatio = 0;           % validation set [%]
net.divideParam.testRatio = 0;          % test set [%]
% train a neural network
[net,tr,Y,E] = train(net,P,T);
% show network
view(net)
```

Solving XOR problem with a multilayer perceptron (3/4)

- Plot targets and network response to see how good the network learns the data

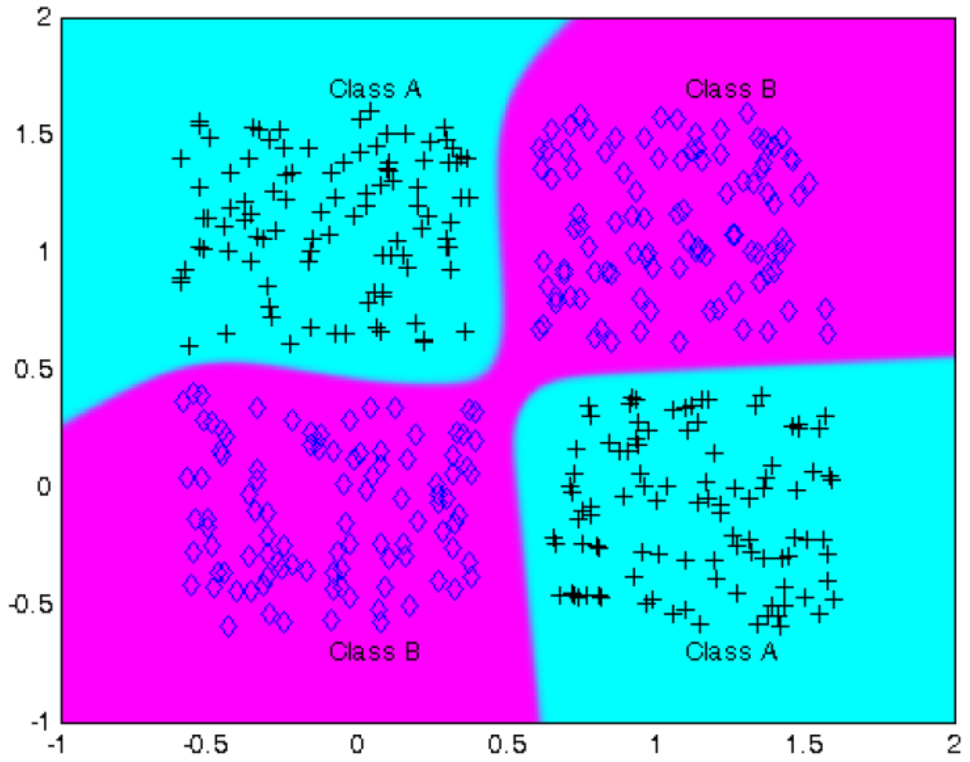
```
figure(2)
plot(T','linewidth',2)
hold on
plot(Y','r--')
grid on
legend('Targets','Network response','location','best')
ylim([-1.25 1.25])
```



Solving XOR problem with a multilayer perceptron (4/4)

- Plot classification result for the complete input space

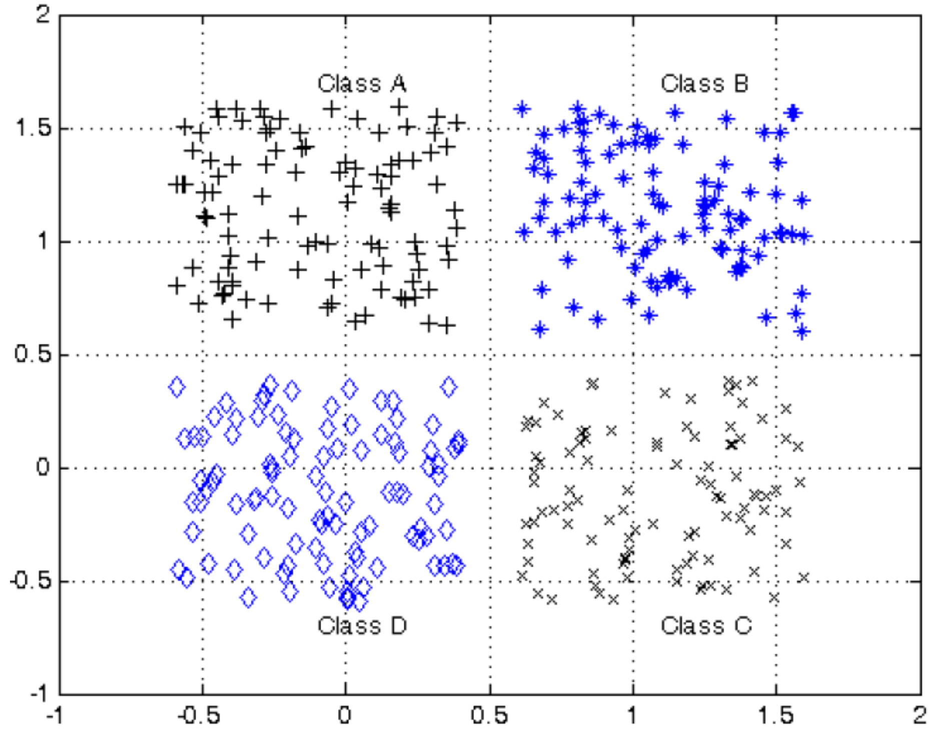
```
generate a grid
span = -1:.005:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simulate neural network on a grid
aa = net(pp);
% translate output into [-1,1]
%aa = -1 + 2*(aa>0);
% plot classification regions
figure(1)
mesh(P1,P2,reshape(aa,length(span),length(span))-5);
colormap cool
```



Classification of a 4-class problem with a multilayer perceptron (1/4)

- Define 4 clusters of input data

```
close all, clear all, clc, format compact
% number of samples of each class
K = 100;
% define classes
q = .6;       % offset of classes
A = [rand(1,K)-q; rand(1,K)+q];
B = [rand(1,K)+q; rand(1,K)+q];
C = [rand(1,K)+q; rand(1,K)-q];
D = [rand(1,K)-q; rand(1,K)-q];
% plot classes
plot(A(1,:),A(2,:), 'k+')
hold on
grid on
plot(B(1,:),B(2,:), 'b*')
plot(C(1,:),C(2,:), 'kx')
plot(D(1,:),D(2,:), 'bd')
% text labels for classes
text(.5-q,.5+2*q,'Class A')
text(.5+q,.5+2*q,'Class B')
text(.5+q,.5-2*q,'Class C')
text(.5-q,.5-2*q,'Class D')
```



Classification of a 4-class problem with a multilayer perceptron (2/4)

- Define output coding for all 4 clusters

```
% coding (+1/-1) of 4 separate classes
a = [-1 -1 -1 +1]';
b = [-1 -1 +1 -1]';
d = [-1 +1 -1 -1]';
c = [+1 -1 -1 -1] ';
```

- Prepare inputs & outputs for network training

```
% define inputs (combine samples from all four classes)
P = [A B C D];
% define targets
T = [repmat(a,1,length(A)) repmat(b,1,length(B)) ...
      repmat(c,1,length(C)) repmat(d,1,length(D)) ];
```

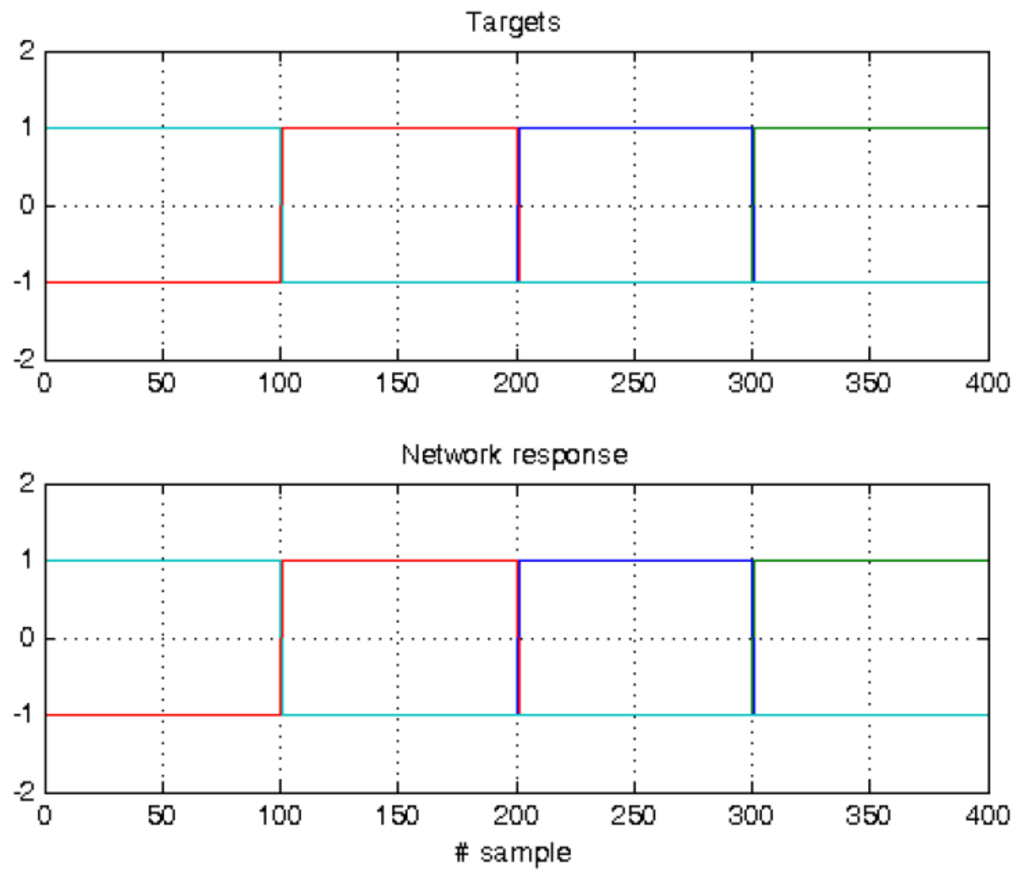
- Create and train a multilayer perceptron

```
% create a neural network
net = feedforwardnet([4 3]);
% train net
net.divideParam.trainRatio = 1;     % training set [%]
net.divideParam.valRatio = 0;       % validation set [%]
net.divideParam.testRatio = 0;      % test set [%]
% train a neural network
[net,tr,Y,E] = train(net,P,T);
% show network
view(net)
```

Classification of a 4-class problem with a multilayer perceptron (3/4)

- Evaluate network performance and plot results

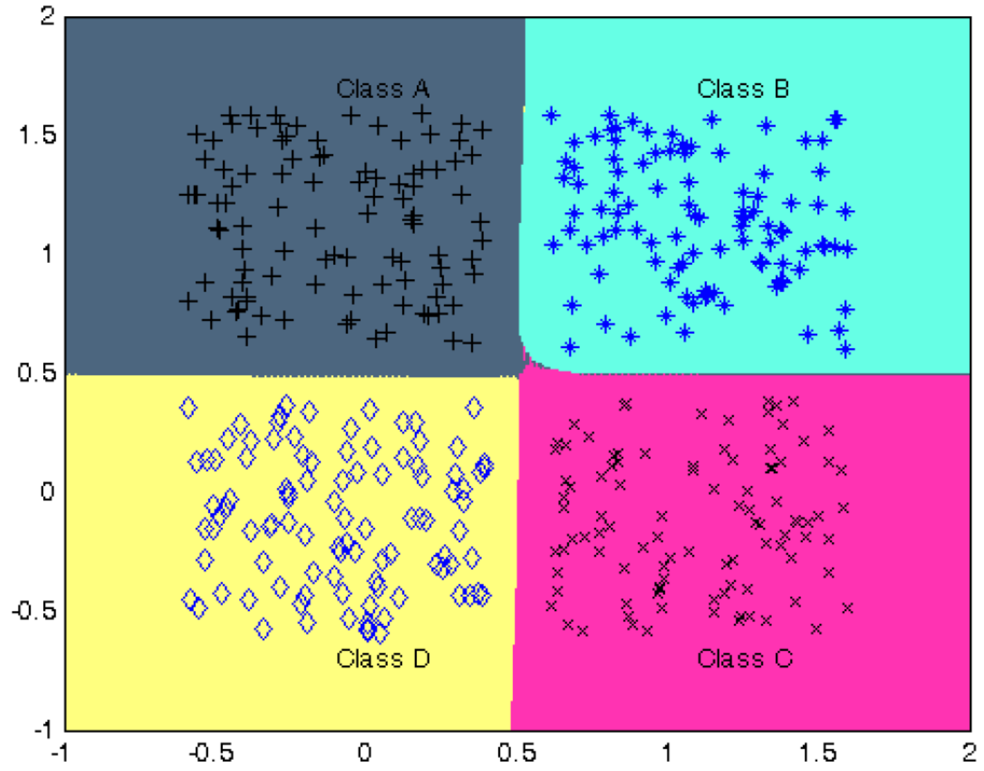
```
% evaluate performance: decoding network response
[m,i] = max(T);             % target class
[m,j] = max(Y);             % predicted class
N = length(Y);              % number of all samples
k = 0;                      % number of missclassified samples
if find(i-j),               % if there exist missclassified samples
   k = length(find(i-j));   % get a number of missclassified samples
end
fprintf('Correct classified samples: %.1f%% samples\n', 100*(N-k)/N)
% plot network output
figure;
subplot(211)
plot(T')
title('Targets')
ylim([-2 2])
grid on
subplot(212)
plot(Y')
title('Network response')
xlabel('# sample')
ylim([-2 2])
grid on
```



Classification of a 4-class problem with a multilayer perceptron (4/4)

- Plot classification result for the complete input space

```
% generate a grid
span = -1:.01:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simualte neural network on a grid
aa = net(pp);
% plot classification regions based on MAX activation
figure(1)
m = mesh(P1,P2,reshape(aa(1,:),length(span),length(span))-5);
set(m,'facecolor',[1 0.2 .7],'linestyle','none');
hold on
m = mesh(P1,P2,reshape(aa(2,:),length(span),length(span))-5);
set(m,'facecolor',[1 1.0 0.5],'linestyle','none');
m = mesh(P1,P2,reshape(aa(3,:),length(span),length(span))-5);
set(m,'facecolor',[.4 1.0 0.9],'linestyle','none');
m = mesh(P1,P2,reshape(aa(4,:),length(span),length(span))-5);
set(m,'facecolor',[.3 .4 0.5],'linestyle','none');
view(2)
```



Exercises Part 2

1. Train and test a multilayer perceptron to perform classification of a 5-class problem.
   Use *dataset_mlp.mat* (X are inputs, T are targets).

2. Plot classification result for the complete input space.

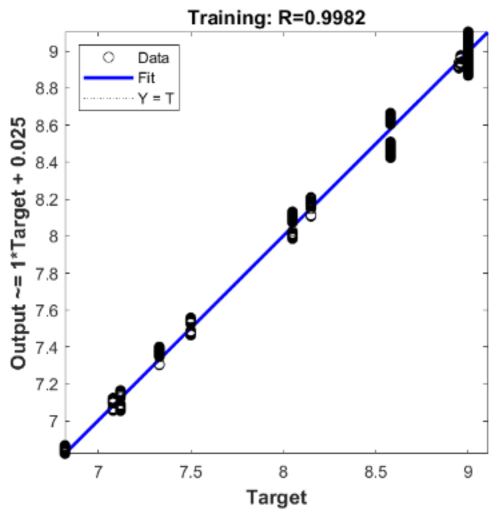
Fitnet

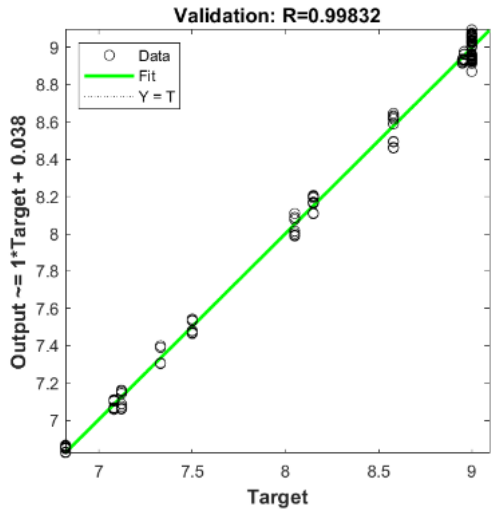
Generate fitting neural network (Fitnet)

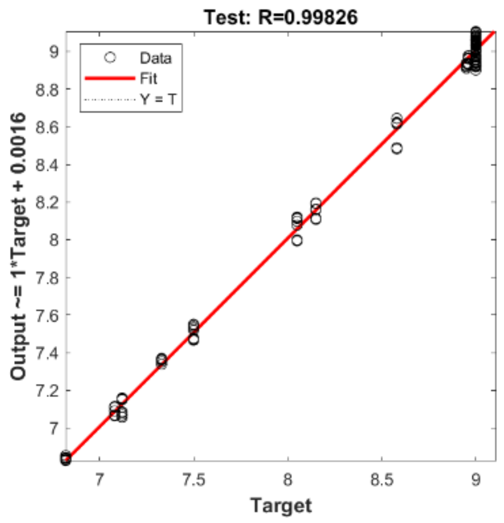
```
net = fitnet(hiddenSizes,trainFcn)

x= inputs;
t = targets;

% Create a Fitting Network
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);

net = train(net, x, t);

output=net(net, x);
```

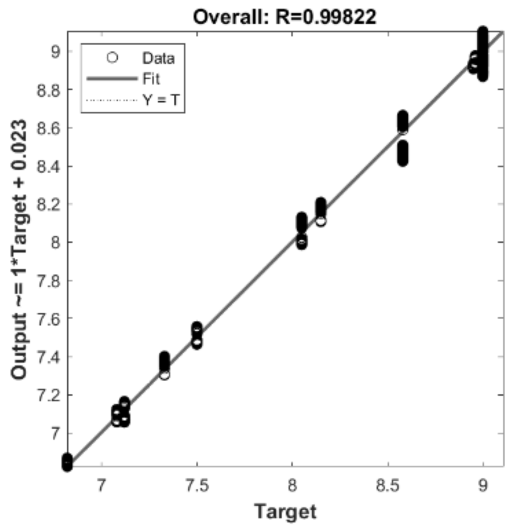
Plot linear regression

plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)









Exercises Part 3

1. Train and test a fitting neural network (Fitnet) using *dataset_regression.mat*.
2. Evaluate the performance by plotting the regression and the error histogram.

Patternnet

Generate pattern recognition network (patternnet)

net = patternnet(hiddenSizes, trainFcn, performFcn)

creates an object named *net* of kind network, representing a 2-layer ANN.

- The number of hidden units has to be provided as a single integer number, expressing the size of the hidden layer, or as an integer row vector, whose elements indicate the size of the correspondent hidden layers.
- The second optional parameter selects the training algorithm by a string saved in the *trainFcn* property, which in the default case takes the value *'trainscg'* (*Scaled Conjugate Gradient Descent* methods).
- The third optional parameter selects the performance function by a string saved in the *performFcn* property, which in the default case takes the value *'crossentropy'*.

The object has several options, which can be reached by the dot notation *object.property* or explore by clicking on the interactively visualization of the object in the *Matlab Command Window*, which allows to see all the available options for each property too.

The network object is still not fully defined, since some variables will be adapted to fit the data dimension at the calling of the function train.

Dataset

Data for ANNs training (as well as for others available Machine Learning methods) must be provided in matrix form, storing each sample column-wise.

For example, data to define the *XOR* problem can be simply defined via an input matrix *X* and a target matrix *Y* as:

```
X = [0 0 1 1; 0 1 0 1];
Y = [0 1 1 0];
```

- For 2-class problem targets can be provided as a row vector of the same length of the number of samples.
- For multi-class problem (and as an alternative for 2-class problem too) targets can be provided in the one-hot encoding form, i.e. as a matrix with as many columns as the number of samples, each one composed by all 0 with only a 1 in the position indicating the class.

ind2vec and **vec2ind** allow indices to be represented either by themselves, or as vectors containing a *1* in the row of the index they represent.

Configuration

Once we have defined data, the network can be fully defined and designed by the command:

net = configure(net , X, Y);

- For each layer, an object of kind *nnetLayer* is created and stored in a cell array under the field *layers* of the *network* object. The number of connections (the weights of the network) for each units corresponds to the layer input dimension.
- The options of each layer can be reached by the dot notation *object.layer{numberOfLayer}.property*.
- The field *initFcn* contains the weights initialization methods.
- The activation function is stored in the *transferFcn* property.
- In the hidden layers the default values is the *'tansig'* (*Hyperbolic Tangent Sigmoid*), whereas the output layers has the *'logsig'* (*Logistic Sigmoid*) or the *'softmax'* for 1-dimensional and multi-dimensional target respectively.
- The 'crossentropy' penalty function is set by default in the field *performFcn*.

Training (1/7)

The function *train* itself makes available many options (as for instance *useParallel* and *useGPU* for heavy computations) directly accessible from its interactive help window. However, it can take as input just the *network* object, the *input* and the *target* matrices.

The optimization starts by dividing data in *Training*, *Validation* and *Test* sets. The splitting ratio can be changed by the options *divideParam*.
In the default setting, data are randomly divided, but if you want for example to decide which data are used for *test*, you can change the way the data are distributed by the option *divideFcn*.

In this case, because of the small size of the dataset, we drop *validation* and *test* by setting:

```
net.divideFcn = ''
```

Training (2/7)

In the following code, we set the training function to the classic *gradient descent method 'traingd'*, we deactivate the training interactive *GUI* by *net.trainParam.showWindow (boolean)* and activate the printing of the training state in the *Command Window* by *net.trainParam.showCommandLine(boolean)*. Also the *learning rate* is part of the *trainParam* options under the fields *lr*.

```
net.trainFcn = 'traingd'
net.trainParam.showWindow = 0
net.trainParam.showCommandLine = 1
net.trainParam.lr = 0.01
```

Training (3/7)

Training starts by the calling:

[net, t r] = train(net, X, Y)

this generates a printing, ending in this case with:

```
Epoch 825/1000, Time 0.426, Performance 0.51544/0, Gradient 0.15234/1e-05, Validation Checks 0/6
Epoch 850/1000, Time 0.439, Performance 0.50987/0, Gradient 0.14626/1e-05, Validation Checks 0/6
Epoch 875/1000, Time 0.452, Performance 0.50472/0, Gradient 0.14073/1e-05, Validation Checks 0/6
Epoch 900/1000, Time 0.467, Performance 0.49994/0, Gradient 0.13569/1e-05, Validation Checks 0/6
Epoch 925/1000, Time 0.48, Performance 0.49549/0, Gradient 0.1311/1e-05, Validation Checks 0/6
Epoch 950/1000, Time 0.493, Performance 0.49133/0, Gradient 0.12693/1e-05, Validation Checks 0/6
Epoch 975/1000, Time 0.506, Performance 0.48742/0, Gradient 0.12313/1e-05, Validation Checks 0/6
Epoch 1000/1000, Time 0.518, Performance 0.48373/0, Gradient 0.11968/1e-05, Validation Checks 0/6
Training with TRAINGD completed: Maximum epoch reached.
```

Training (4/7)

The training stops after the max number of epochs is reached (which can be set by options *object.trainParam.epochs*).
Each column shows the state of one of the stopping criteria used.
The output variable *tr* stores the training options.
The fields *perf*, *vperf* and *tperf* contain the performance of the network evaluated at each epoch on the *Training*, *Validation* and *Test* sets respectively (the last two are *NaN* in this case), which can be used for example to plot performances.

If we pass data organized in a single matrix, the function will exploit the full batch learning method accumulating gradients overall the training set.

Training (5/7)

To set a mini-batch mode, data have to be manually split in sub-matrix with the same number of column and organized in a *cell-array*. However, let us consider for a moment a general data set composed by *N* samples in the features space $\mathbb{R}^D$ with a target of dimension *C*, so that $\mathrm{X} \in \mathbb{R}^{D \times N}$ and $\mathrm{Y} \in \mathbb{R}^{C \times N}$. All the mini-batches have to be of the same size *b*, so that it is in general convenient to choose the batch size to be a factor of *N*.
In this case, we can generate data for the training function organizing the input and target in the correspondent *cell-array*.

Training (6/7)

```
N = size(X, 2 ) ;                    % number of samples
n_batch = N/ batchsize;              % number of batches

input{n_batch} = [ ] ;               % input cell –array initialization
target{n_batch} = [ ] ;              % target cell –array initialization
p = randperm(N) ;                    % generating a random permutated index for data shuffling
X = X( : , p) ;                      % samples permutation
Y = Y( : , p) ;                      % target permutation
for i  = 1 : n_batch
   input{i} = X(:, (1 : batchsize) + (i –1) * batchsize) ;
   target{i} = Y(:, (1 : batchsize) + (i –1) * batchsize);
end
```

Training (7/7)

However, in order to perform a pure Stochastic Gradient Descent optimization, in which the ANNs parameters are updated for each sample, the training function '*train*' has to be employed skipping to split data previously. A remark has to be done since this particular function does not support the GPU computing.
The network (trained or not) can be easily evaluated on data by passing the input data as argument to a function named as the network object. Performance of the prediction with respect to the targets can be evaluated by the function perform according to the correspondent loss function option *object.performFcn*:

```
f = net(X);
perform(net, Y, f)
```

Stopping criteria and Regularization (1/2)

Early stopping is a well known procedure in Machine Learning to avoid overfitting and improve generalization. Different kind of stopping criteria can be used regulated by the network object options in the field *trainParam*.
Arbitrarily methods are based on the number of epochs (*epochs*) and the training time (*time*, default *Inf*).

A criterion based on the training set:

- check the *loss* (*object.trainParam.goal, default = 0*) or
- the parameters gradients (*object.trainParam.min_grad, default = 10−6*) to reach a minimum threshold.

Stopping criteria and Regularization (2/2)

A general early stopping method is implemented by checking the error on the validation set and interrupting training when validation error does not improve for a number of consecutive epochs given by *max_fail (default=6)*.

Further regularization methods can be configured by the property *performParam* of the network object. The field *regularization* contains the weight (real number in [0, 1]) balancing the contribution of a term trying to minimizing the norm of the network weights versus the satisfaction of the penalty function. However, the network is designed to mainly rely on the validation checks, indeed regularization applies only to few kind of penalties and the default weight is 0.

Drawing separation surfaces (1/3)

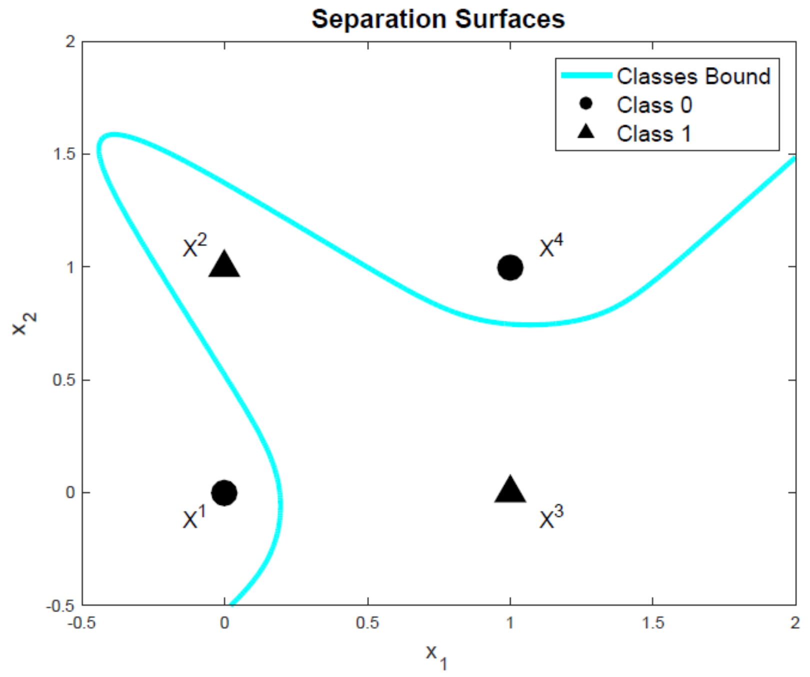
When dealing with low dimensional data (as in the *XOR* case), can be useful to visualize the prediction of the network directly in the input space.

The network predictions will be evaluated on a grid of the input space, generated by the Matlab function *meshgrid*, since the main functions used for the plotting (*contour* or, if you want a color surface *pcolor*) require as input three matrices of the same dimensions expressing, in each correspondent element, the coordinates of a 3-D point (which in our case will be first input dimension, second input dimension and prediction).

Drawing separation surfaces (2/3)

Once we trained the network described until now, the boundary for the 2-classes separation is generated by the following code

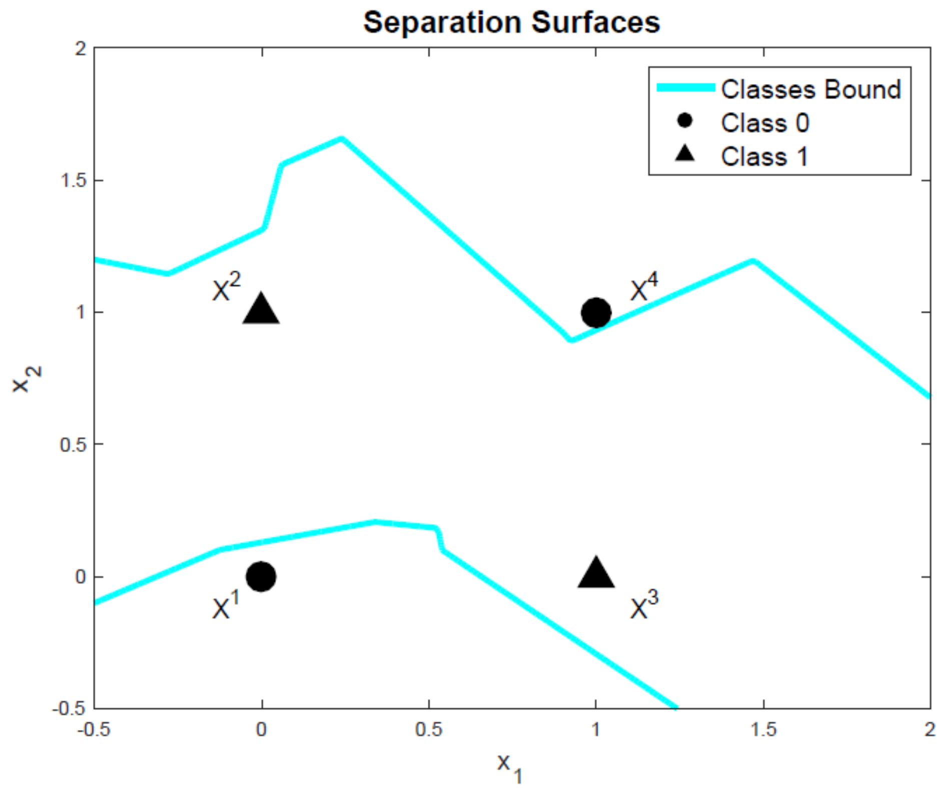
```
% generating input space grid
[xp1, xp2] = meshgrid(−0.5 : 0.01 : 2, −0.5 : 0.01 : 2);
f = net([xp1(:)'; xp2(:)']);                        % network evaluation on the gridding (reshaped to fit network input dimension)
f = reshape(f, size(xp1, 1), []);                    % reshaping prediction in correspondent matrix form
contour(xp1, xp2, f, [.5, .5], 'LineWidth', 3 , 'Color', 'c');   % drawing separation surfaces
hold on;
scatter(X(1, [1, 4]), X(2, [1, 4] ), 200, 'o', 'filled', 'MarkerEdgeColor', 'k', ...
'MarkerFaceColor', 'k', 'LineWidth', 2) ;            % drawing data points
scatter(X(1, [2, 3]), X(2, [2, 3]), 200, '^', 'filled', 'MarkerEdgeColor', 'k', ...
'MarkerFaceColor', 'k', 'LineWidth', 2);
axis([−0.5, 2, −0.5, 2]);                            %setting axis bounds
%labeling data points
c={'X^1', 'X^2', 'X^3', 'X^4'};                      %labels
dx = [−.15, −.15, .1, .1];                           %labels horizontal translation wrt points
25dy=[−.1, .1, −.1, 1];                              %labels vertical translation wrt points
text(X(1, :) + dx, X(2, :) + dy, c, 'FontSize', 14); %showing labels as text
%plotlabels
xlabel('x_1', 'FontSize', 14)
ylabel('x_2', 'FontSize', 14)
title('SeparationSurfaces', 'FontSize', 16);
h = legend({'ClassesBound', 'Class0', 'Class1'}, 'Location', 'NorthEast');
set(h, 'FontSize', 14);
```



Drawing separation surfaces (3/3)

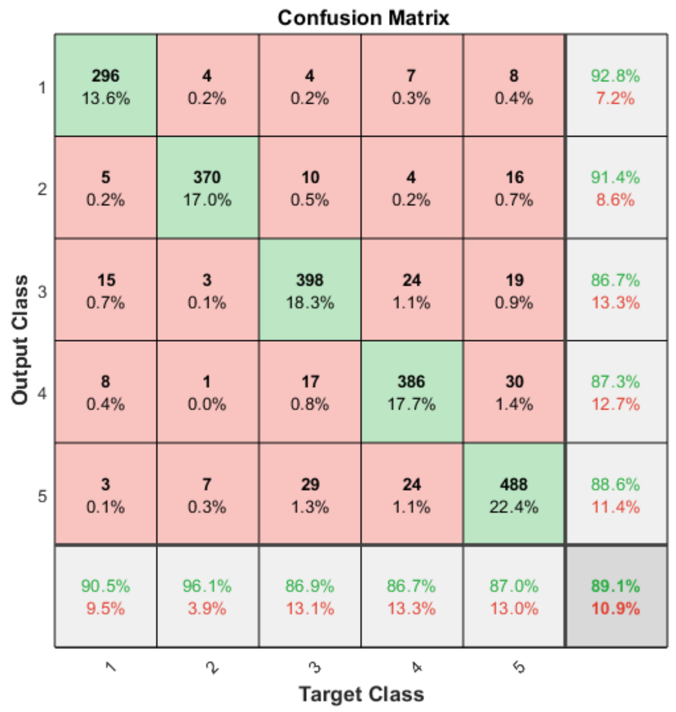
We report the same evaluation after the training of a 4-layers network using 5, 3 and 2 units in the first, second and third hidden layers respectively, each one using the *ReLU* as activation (*'poslin'* in Matlab). This new network can be defined by:

```
% 3+output layers network initialization
n = patternnet([5, 3, 2]);
% network configuration
net = configure(n, X, Y);
% setting optimization function
net.trainFcn = 'traingd';
% setting data splitting ratios (illustrative)
net.divideParam.trainRatio = 1;
net.divideParam.valRat io = 0;
net.divideParam.testRatio = 0;
net.trainParam.showCommandLine = 1;
net.trainParam.lr = 0.01;
% setting the activation layer−wise
net.layers{1}.transferFcn = 'poslin';
net.layers{2}.transferFcn = 'poslin';
net.layers{3}.transferFcn = 'poslin';
```



Plot classification confusion matrix

plotconfusion(targets1,outputs1, 'name1',targets2,outputs2, 'name2',...)

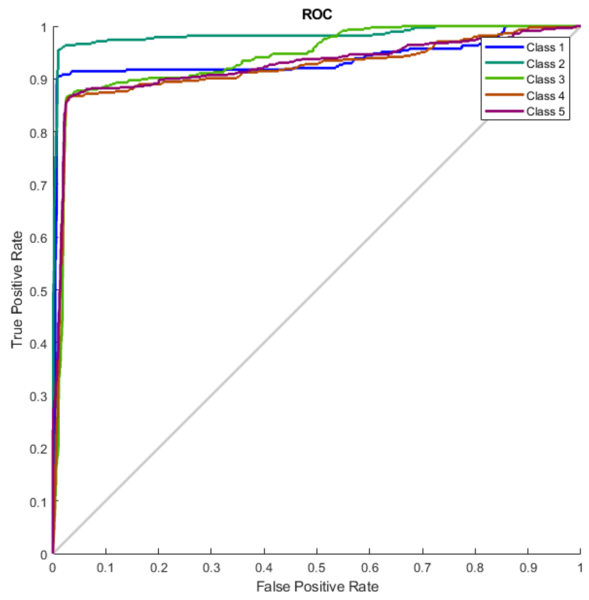


Confusion Matrix

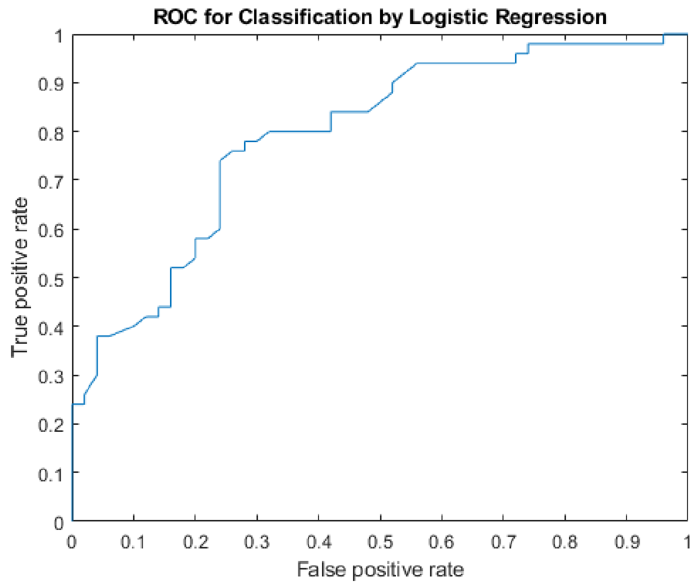
Output Class \ Target Class	1	2	3	4	5	
1	296 13.6%	4 0.2%	4 0.2%	7 0.3%	8 0.4%	92.8% 7.2%
2	5 0.2%	370 17.0%	10 0.5%	4 0.2%	16 0.7%	91.4% 8.6%
3	15 0.7%	3 0.1%	398 18.3%	24 1.1%	19 0.9%	86.7% 13.3%
4	8 0.4%	1 0.0%	17 0.8%	386 17.7%	30 1.4%	87.3% 12.7%
5	3 0.1%	7 0.3%	29 1.3%	24 1.1%	488 22.4%	88.6% 11.4%
	90.5% 9.5%	96.1% 3.9%	86.9% 13.1%	86.7% 13.3%	87.0% 13.0%	89.1% 10.9%

Plot receiver operating characteristic

plotroc(targs1,outs1,'name1',targs2,outs2,'name2',...)



[X,Y,T,AUC] = perfcurve(labels,scores,posclass)



scores: a vector of classifier predictions
labels: true class labels
posclass: positive class label.

Hyperparameters

- Number of neurons
- Number of hidden layers
- Activation functions
- Learning Rate
- Momentum
- Performance function

```
num_neurons = 5 : 1 : 15;
for i = 1 : numel(num_neurons)
   k = num_neurons(i);
   …
end
```

Exercises Part 4

1. Train and test a pattern recognition neural network(Patternnet) using *dataset_classification.mat*.

2. Evaluate the performance by plotting the confusion matrix and ROC.
   Using the command `[c,cm]=confusion(testT,testY)` compute the overall percentages of correct and incorrect classification

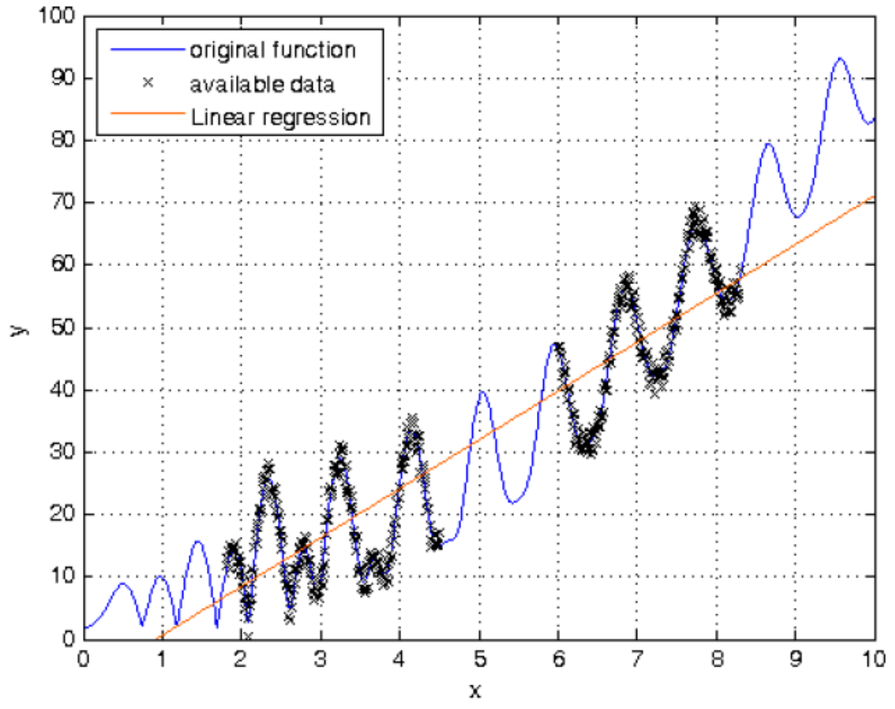
Function approximation with RBFN

Data generator

```
%% Data generator function
function [X,Xtrain,Ytrain,fig] = data_generator()
   % data generator
   X = 0.01:.01:10;
   f = abs(besselj(2,X*7).*asind(X/2) + (X.^1.95)) + 2;
   fig = figure;
   plot(X,f,'b-')
   hold on
   grid on
   % available data points
   Ytrain = f + 5*(rand(1,length(f))-.5);
   Xtrain = X([181:450 601:830]);
   Ytrain = Ytrain([181:450 601:830]);
   plot(Xtrain,Ytrain,'kx')
   xlabel('x')
   ylabel('y')
   ylim([0 100])
   legend('original function','available data','location','northwest')
end
```

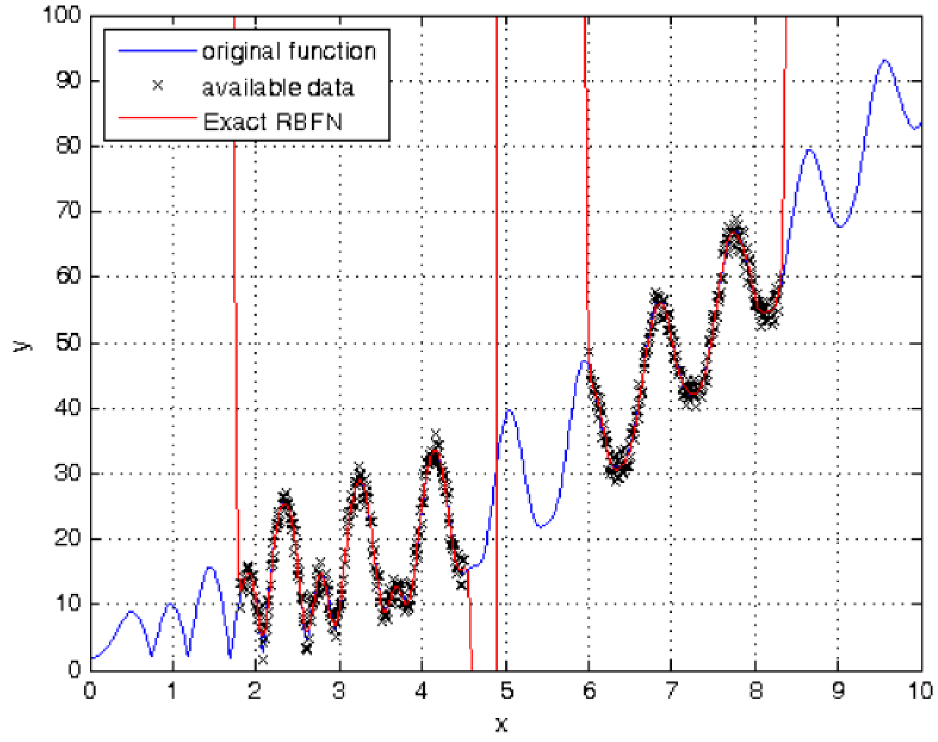
Linear Regression

```
close all, clear all, clc, format compact
% generate data
[X,Xtrain,Ytrain,fig] = data_generator();

%--------------------------------
% no hidden layers
net = feedforwardnet([]);
% one hidden layer with linear transfer functions
% net = feedforwardnet([10]);
% net.layers{1}.transferFcn = 'purelin';
% set early stopping parameters
net.divideParam.trainRatio = 1.0;    % training set [%]
net.divideParam.valRatio = 0.0;      % validation set [%]
net.divideParam.testRatio = 0.0;     % test set [%]
% train a neural network
net.trainParam.epochs = 200;
net = train(net,Xtrain,Ytrain);
%--------------------------------
% view net
view (net)
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'color',[1 .4 0])
legend('original function','available data','Linear regression','location','northwest')
```



Exact RBFN

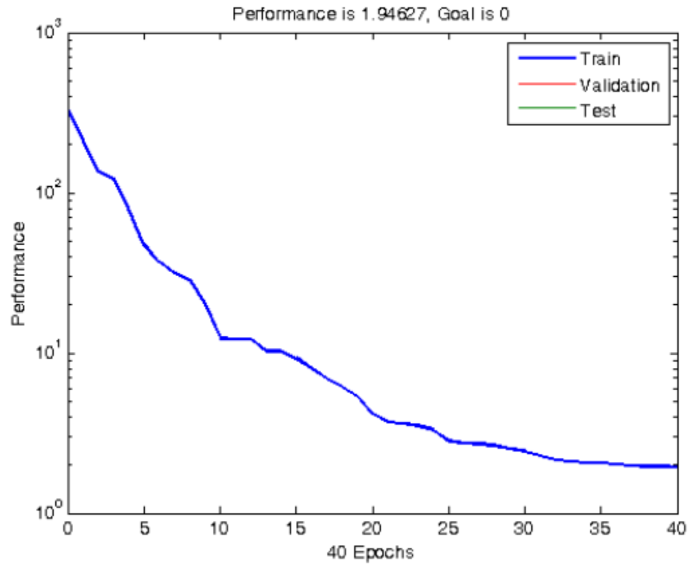
```
% generate data
[X,Xtrain,Ytrain,fig] = data_generator();
%--------------------------------
% choose a spread constant
spread = .4;
% create a neural network
net = newrbe(Xtrain,Ytrain,spread);
%--------------------------------
% view net
view (net)
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'r')
legend('original function','available data','Exact RBFN','location','northwest')
```

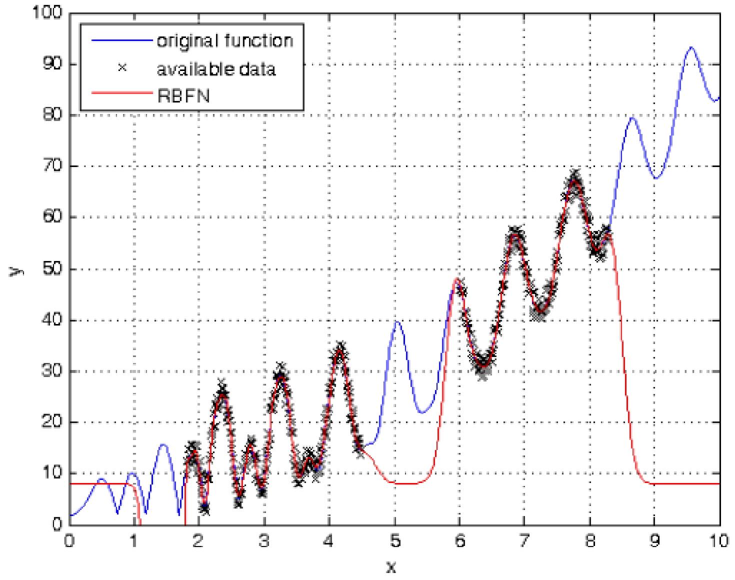


RBFN

```
% generate data
[X,Xtrain,Ytrain,fig] = data_generator();
%---------------------------------
% choose a spread constant
spread = .2;
% choose max number of neurons
K = 40;
% performance goal (SSE)
goal = 0;
% number of neurons to add between displays
Ki = 5;
% create a neural network
net = newrb(Xtrain,Ytrain,goal,spread,K,Ki);
%---------------------------------
% view net
view (net)
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'r')
legend('original function','available data','RBFN','location','northwest')
```

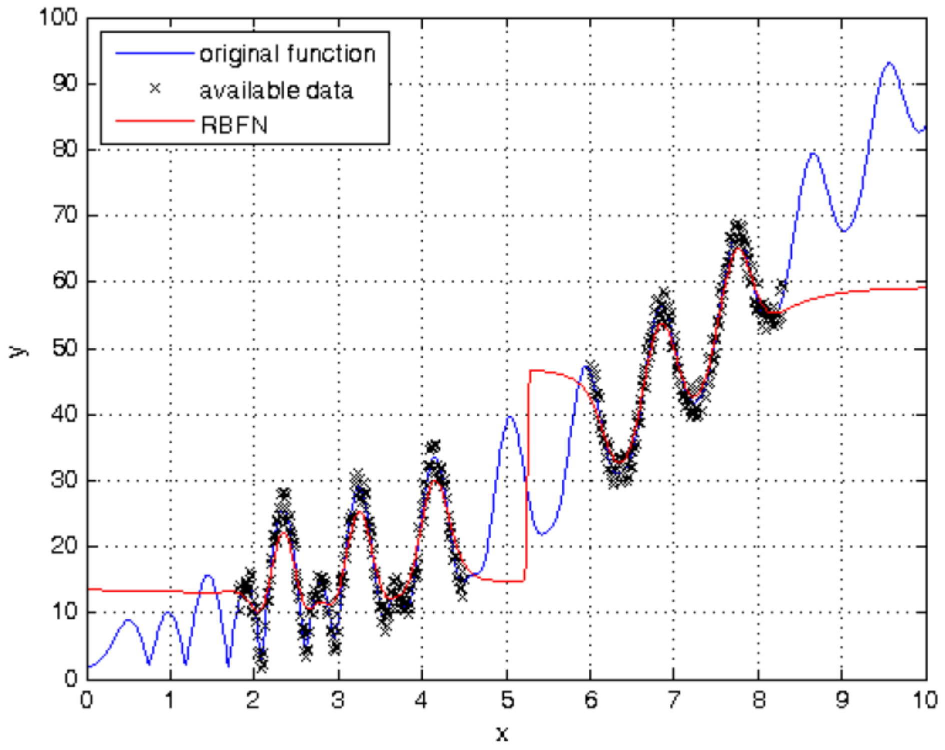
```
NEWRB, neurons = 0, MSE = 333.938
NEWRB, neurons = 5, MSE = 47.271
NEWRB, neurons = 10, MSE = 12.3371
NEWRB, neurons = 15, MSE = 9.26908
NEWRB, neurons = 20, MSE = 4.16992
NEWRB, neurons = 25, MSE = 2.82444
NEWRB, neurons = 30, MSE = 2.43353
NEWRB, neurons = 35, MSE = 2.06149
NEWRB, neurons = 40, MSE = 1.94627
```





Generalized Regression Neural Networks (GRNN)

```
% generate data
[X,Xtrain,Ytrain,fig] = data_generator();
%--------------------------------
% choose a spread constant
spread = .12;
% create a neural network
net = newgrnn(Xtrain,Ytrain,spread);
%--------------------------------
% view net
view (net)
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'r')
legend('original function','available data','RBFN','location','northwest')
```

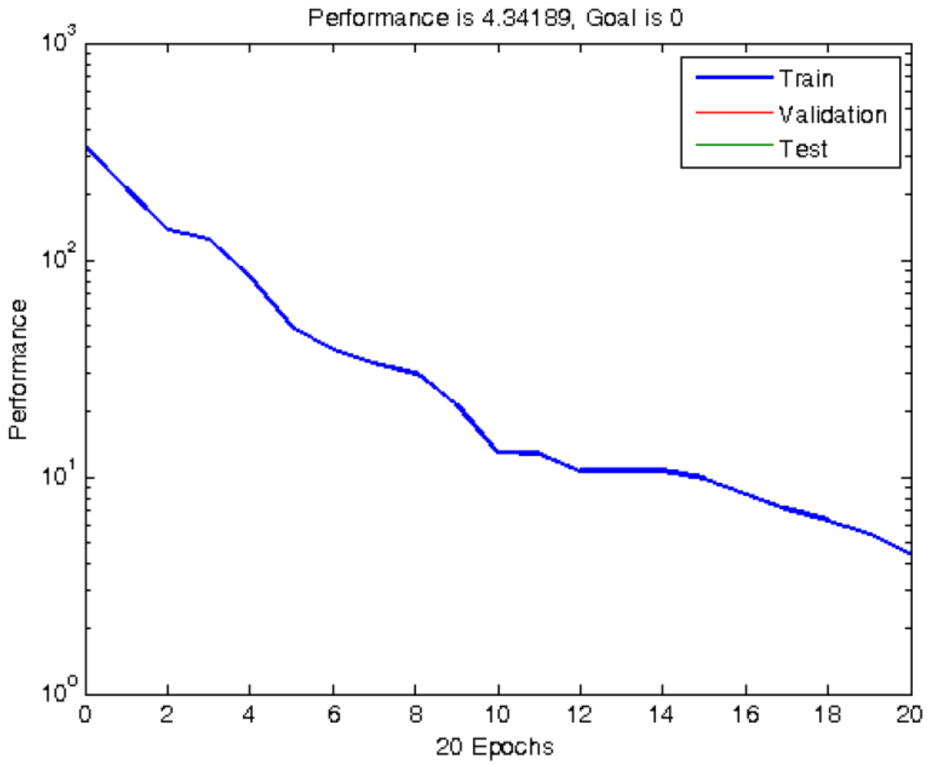


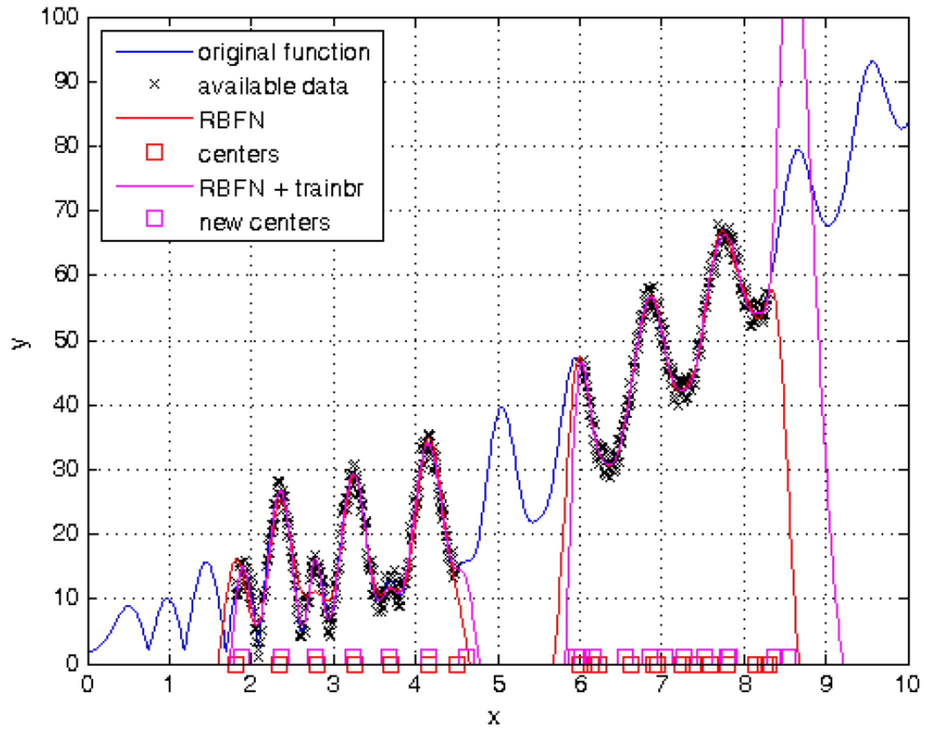
RBFN trained by Bayesian regularization (1/2)

```
generate data
[X,Xtrain,Ytrain,fig] = data_generator();
%--------- RBFN ------------------
% choose a spread constant
spread = .2;
% choose max number of neurons
K = 20;
% performance goal (SSE)
goal = 0;
% number of neurons to add between displays
Ki = 20;
% create a neural network
net = newrb(Xtrain,Ytrain,goal,spread,K,Ki);
%----------------------------------
% view net
view (net)
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'r')
% Show RBFN centers
c = net.iw{1};
plot(c,zeros(size(c)),'rs')
legend('original function','available data','RBFN','centers','location','northwest')
%--------- trainbr ---------------
% Retrain a RBFN using Bayesian regularization backpropagation
net.trainFcn='trainbr';
net.trainParam.epochs = 100;
% perform Levenberg-Marquardt training with Bayesian regularization
net = train(net,Xtrain,Ytrain);
%----------------------------------
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'m')
% Show RBFN centers
c = net.iw{1};
plot(c,ones(size(c)),'ms')
legend('original function','available data','RBFN','centers','RBFN + trainbr',’new centers','location','northwest')
```

RBFN trained by Bayesian regularization (2/2)

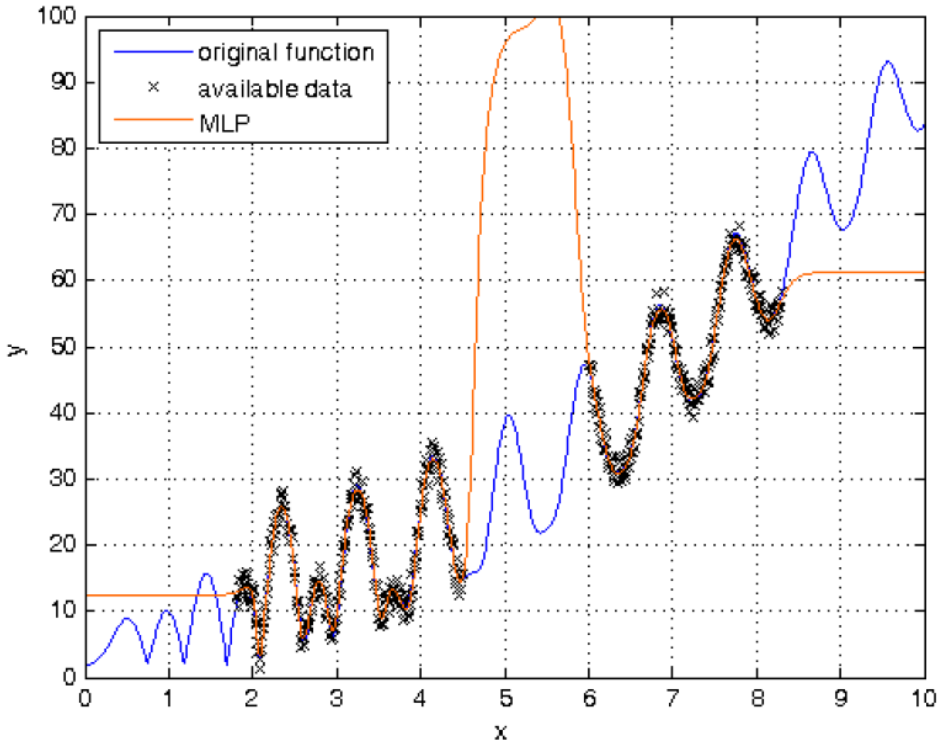
```
NEWRB, neurons = 0, MSE = 334.852
NEWRB, neurons = 20, MSE = 4.34189
```





MLP

```
% generate data
[X,Xtrain,Ytrain,fig] = data_generator();
%--------------------------------
% create a neural network
net = feedforwardnet([12 6]);
% set early stopping parameters
net.divideParam.trainRatio = 1.0;          % training set [%]
net.divideParam.valRatio = 0.0;            % validation set [%]
net.divideParam.testRatio = 0.0;           % test set [%]
% train a neural network
net.trainParam.epochs = 200;
net = train(net,Xtrain,Ytrain);
%--------------------------------
% view net
view (net)
% simulate a network over complete input range
Y = net(X);
% plot network response
figure(fig)
plot(X,Y,'color',[1 .4 0])
legend('original function','available data','MLP','location','northwest')
```



Exercises Part 5

1. Train and test a Radial Basis Neural Networks (RBFN) with the following inputs and outputs
   ```
   X = -1:.1:1;
   T = [-.9602 -.5770 -.0729  .3771  .6405  .6600  .4609 ...
        .1336 -.2013 -.4344 -.5000 -.3930 -.1647  .0988 ...
        .3072  .3960  .3449  .1816 -.0312 -.2189 -.3201];
   ```
   and `eg=0.02(sum-squared error goal)` and `sc=1(spread constant)`

2. Repeat the exercise *1* considering the case of underlapping Neurons `(eg=0.02 and sc=0.01)`

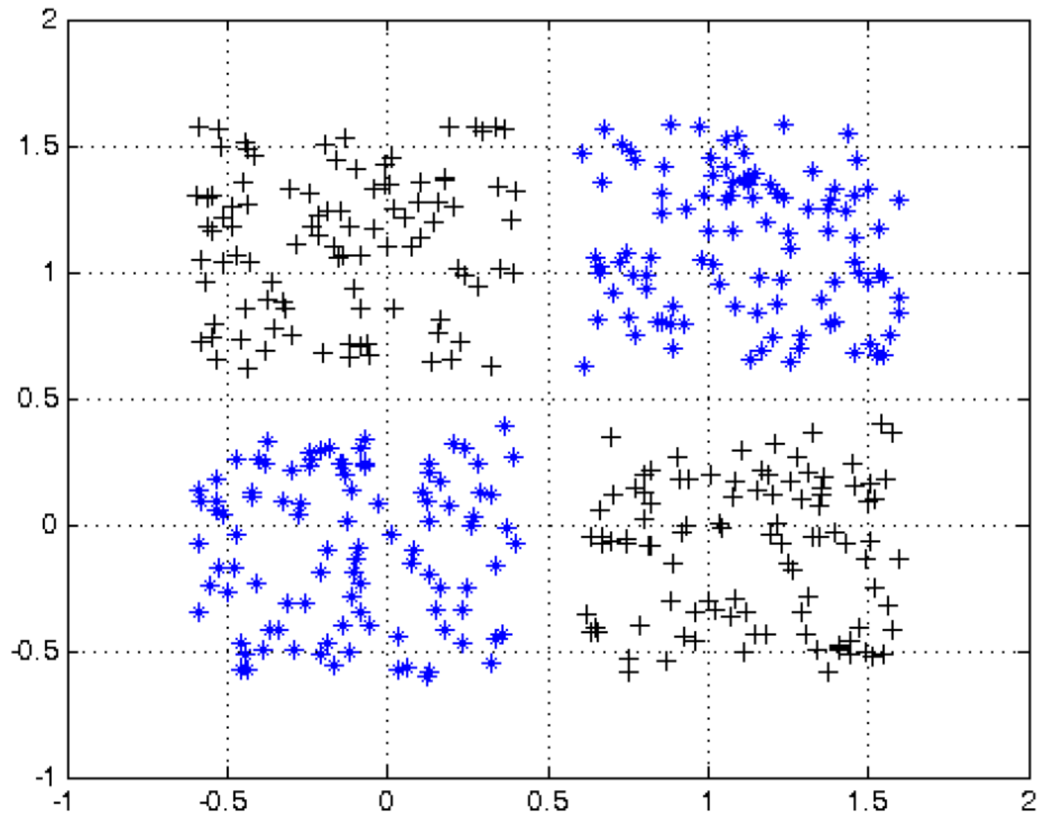
3. Repeat the exercise *1* considering the case of overlapping Neurons `(eg=0.02 and sc=100)`

Radial Basis Function Networks for Classification of XOR problem

Classification of XOR problem with an exact RBFN (1/4)

- Create input data

```
close all, clear all, clc, format compact
% number of samples of each class
K = 100;
% define classes
q = .6;      % offset of classes
% define 2 groups of input data
A = [rand(1,K)-q rand(1,K)+q;
    rand(1,K)+q rand(1,K)-q];
B = [rand(1,K)+q rand(1,K)-q;
    rand(1,K)+q rand(1,K)-q];
% plot data
plot(A(1,:),A(2,:),'k+',B(1,:),B(2,:),'b*')
grid on
hold on
```



Classification of XOR problem with an exact RBFN (2/4)

- Define output coding

```
% coding (+1/-1) for 2-class XOR problem
a = -1;
b = 1;
```

- Prepare inputs & outputs for network training

```
% define inputs (combine samples from all four classes)
P = [A B];
% define targets
T = [repmat(a,1,length(A)) repmat(b,1,length(B))];
```

- Create an exact RBFN

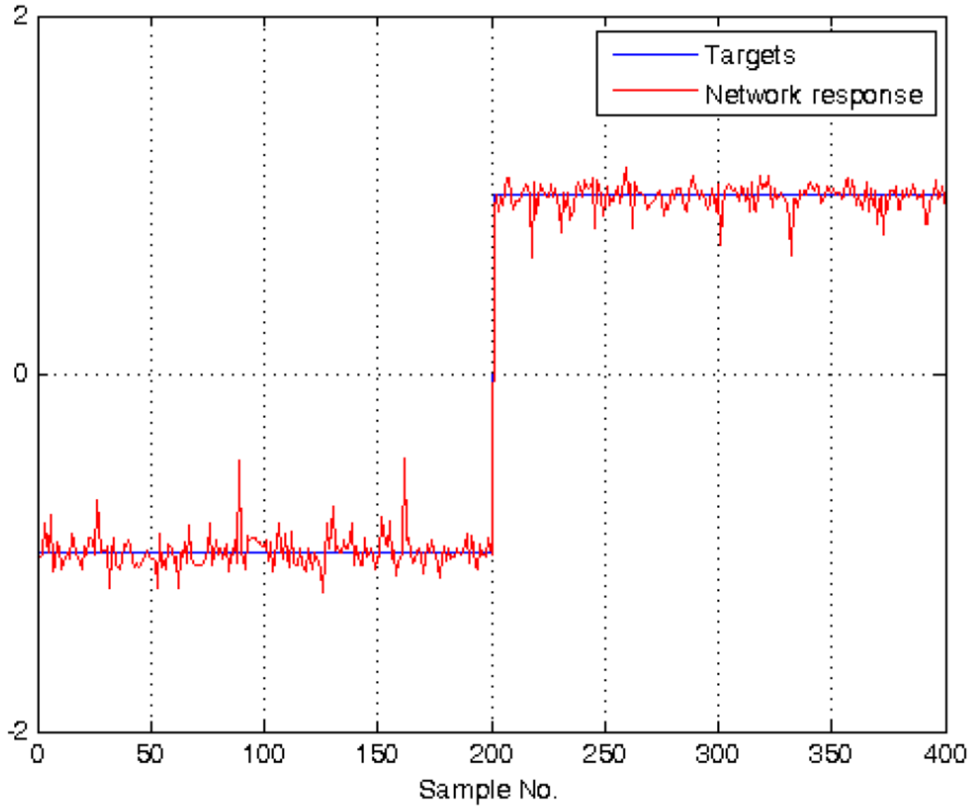
```
% choose a spread constant
spread = 1;
% create a neural network
net = newrbe(P,T,spread);
% view network
view(net)
```

Classification of XOR problem with an exact RBFN (3/4)

- Evaluate network performance

```
% simulate a network on training data
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
fprintf('\nSpread = %.2f\n',spread)
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
figure;
plot(T')
hold on
grid on
plot(Y','r')
ylim([-2 2])
set(gca,'ytick',[-2 0 2])
legend('Targets','Network response')
xlabel('Sample No.’)
```

```
Spread = 1.00
Num of neurons = 400
Correct class = 100.00 %
```



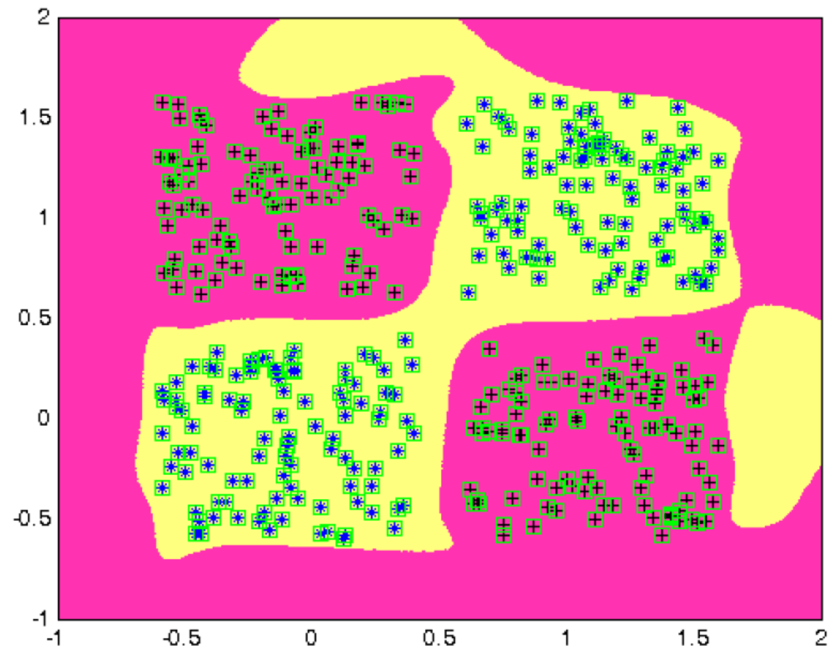
Classification of XOR problem with an exact RBFN (4/4)

- Plot classification result

```
% generate a grid
span = -1:.025:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simualte neural network on a grid
aa = sim(net,pp);
% plot classification regions based on MAX activation
figure(1)
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
view(2)
```

- Plot RBFN centers

```
plot(net.iw{1}(:,1),net.iw{1}(:,2),'gs')
```

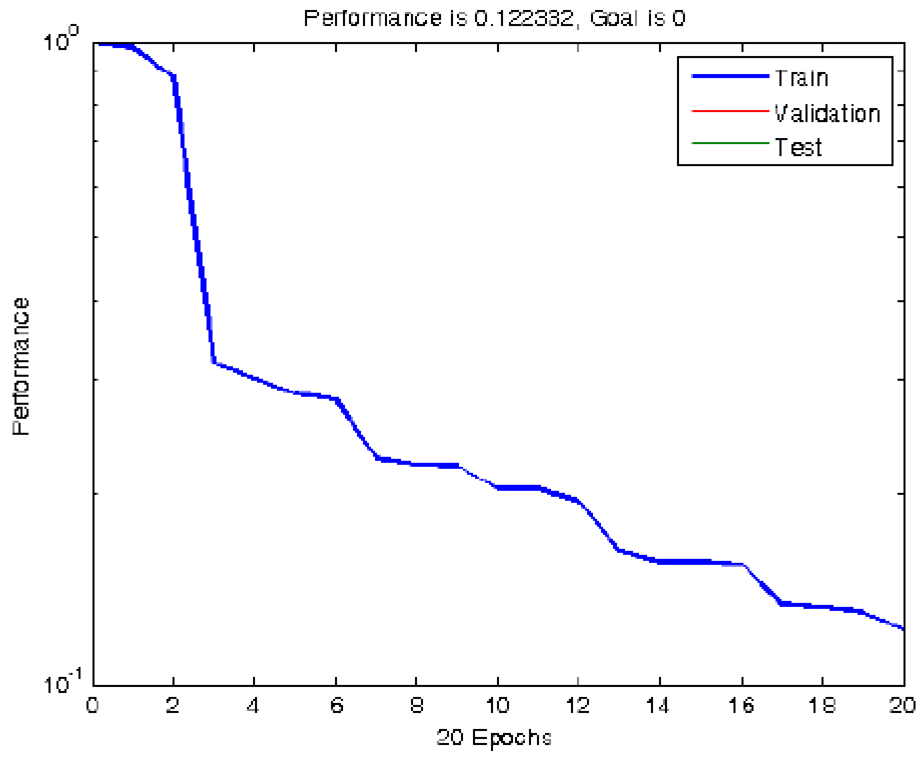


Classification of XOR problem with a RBFN (1/3)

- Create a RBFN

```
% NEWRB algorithm
% The following steps are repeated until the network's mean squared error
% falls below goal:
% 1. The network is simulated
% 2. The input vector with the greatest error is found
% 3. A radbas neuron is added with weights equal to that vector
% 4. The purelin layer weights are redesigned to minimize error
% choose a spread constant
spread = 2;
% choose max number of neurons
K = 20;
% performance goal (SSE)
goal = 0;
% number of neurons to add between displays
Ki = 4;
% create a neural network
net = newrb(P,T,goal,spread,K,Ki);
% view network
view(net)
```

```
NEWRB, neurons = 0, MSE = 1
NEWRB, neurons = 4, MSE = 0.302296
NEWRB, neurons = 8, MSE = 0.221059
NEWRB, neurons = 12, MSE = 0.193983
NEWRB, neurons = 16, MSE = 0.154859
NEWRB, neurons = 20, MSE = 0.122332
```

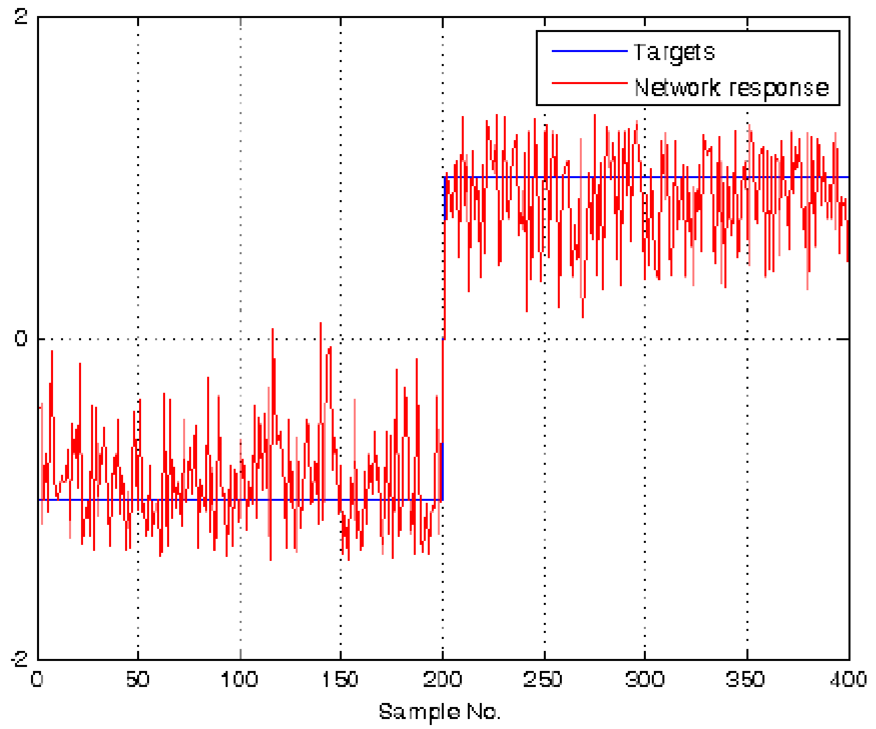


Classification of XOR problem with a RBFN (2/3)

- Evaluate network performance

```
% simulate RBFN on training data
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
fprintf('\nSpread = %.2f\n',spread)
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
figure;
plot(T')
hold on
grid on
plot(Y','r')
ylim([-2 2])
set(gca,'ytick',[-2 0 2])
legend('Targets','Network response')
xlabel('Sample No.')
```

```
Spread = 2.00
Num of neurons = 20
Correct class = 99.50 %
```



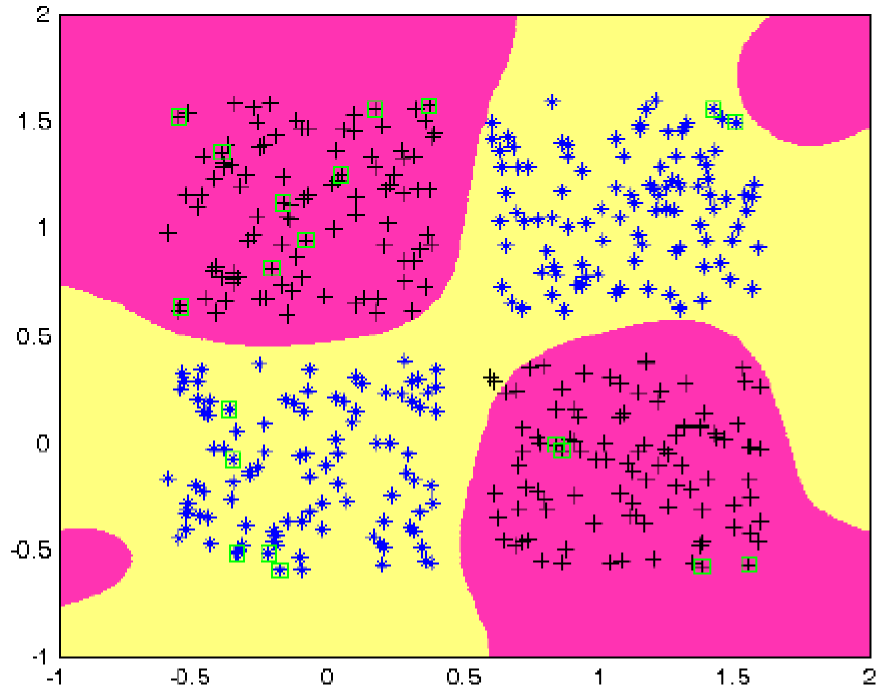
Classification of XOR problem with a RBFN (3/3)

- Plot classification result

```
% generate a grid
span = -1:.025:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simualte neural network on a grid
aa = sim(net,pp);
% plot classification regions based on MAX activation
figure(1)
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
view(2)
```

- Plot RBFN centers

```
plot(net.iw{1}(:,1),net.iw{1}(:,2),'gs')
```



Exercises Part 6

1. Train and test a Radial Basis Function Networks to perform classification of a 5-class problem. Use *dataset_mlp.mat* (X are inputs, T are targets).

2. Plot classification result for the complete input space.

3. Repeat the exercise *1* considering the case of underlapping Neurons and overlapping Neurons

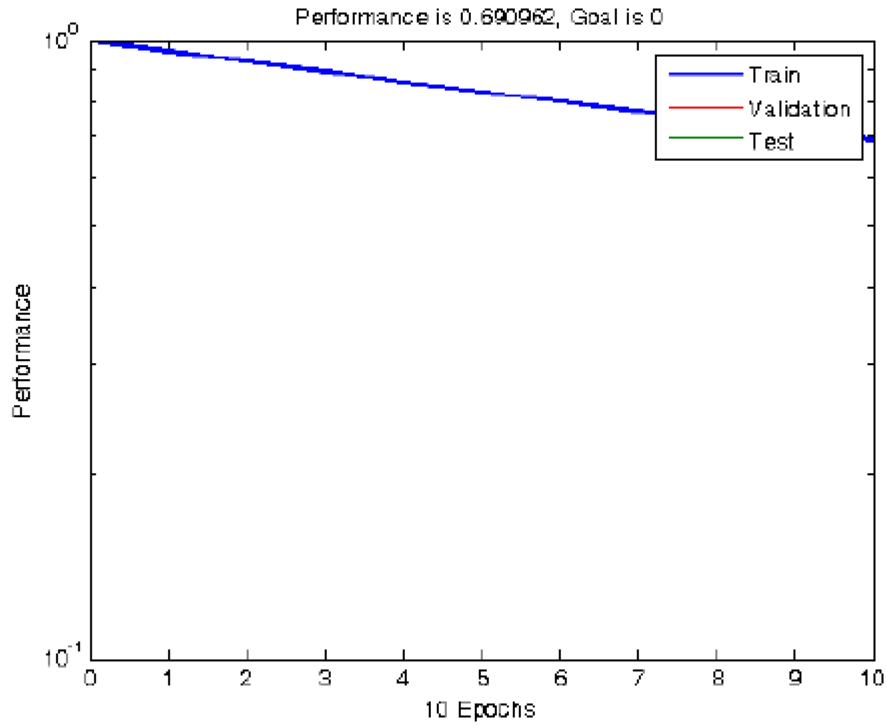
Bayesian regularization for RBFN

Bayesian regularization for RBFN (1/5)

- Create a RBFN

```
% NEWRB algorithm
% The following steps are repeated until the network's mean squared error
% falls below goal:
% 1. The network is simulated
% 2. The input vector with the greatest error is found
% 3. A radbas neuron is added with weights equal to that vector
% 4. The purelin layer weights are redesigned to minimize error
% choose a spread constant
spread = .1;
% choose max number of neurons
K = 10;
% performance goal (SSE)
goal = 0;
% number of neurons to add between displays
Ki = 2;
% create a neural network
net = newrb(P,T,goal,spread,K,Ki);
% view network
view(net)
```

```
NEWRB, neurons = 0, MSE = 1
NEWRB, neurons = 2, MSE = 0.928277
NEWRB, neurons = 4, MSE = 0.855829
NEWRB, neurons = 6, MSE = 0.798564
NEWRB, neurons = 8, MSE = 0.742854
NEWRB, neurons = 10, MSE = 0.690962
```

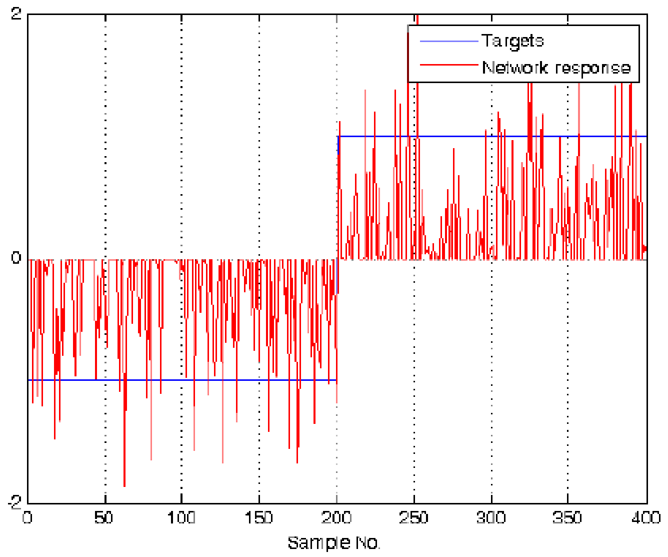


Bayesian regularization for RBFN (2/5)

- Evaluate network performance

```
% check RBFN spread
actual_spread = net.b{1}
% simulate RBFN on training data
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
fprintf('\nSpread = %.2f\n',spread)
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
% to see how good the network learns the data
figure;
plot(T')
ylim([-2 2])
set(gca,'ytick',[-2 0 2])
hold on
grid on
plot(Y','r')
legend('Targets','Network response')
xlabel('Sample No.')
```

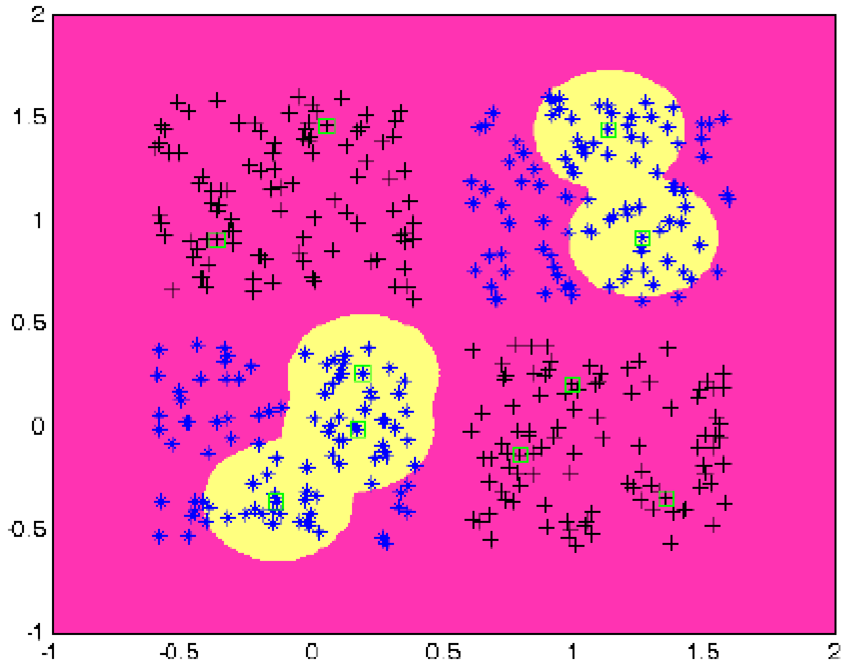
```
actual_spread =
8.3255
8.3255
8.3255
8.3255
8.3255
8.3255
8.3255
8.3255
8.3255
8.3255
Spread = 0.10
Num of neurons = 10
Correct class = 79.50 %
```



Bayesian regularization for RBFN (3/5)

- Plot classification result

```
% generate a grid
span = -1:.025:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simualte neural network on a grid
aa = sim(net,pp);
% plot classification regions based on MAX activation
figure(1)
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
view(2)

% plot RBFN centers
plot(net.iw{1}(:,1),net.iw{1}(:,2),'gs')
```



Bayesian regularization for RBFN (4/5)

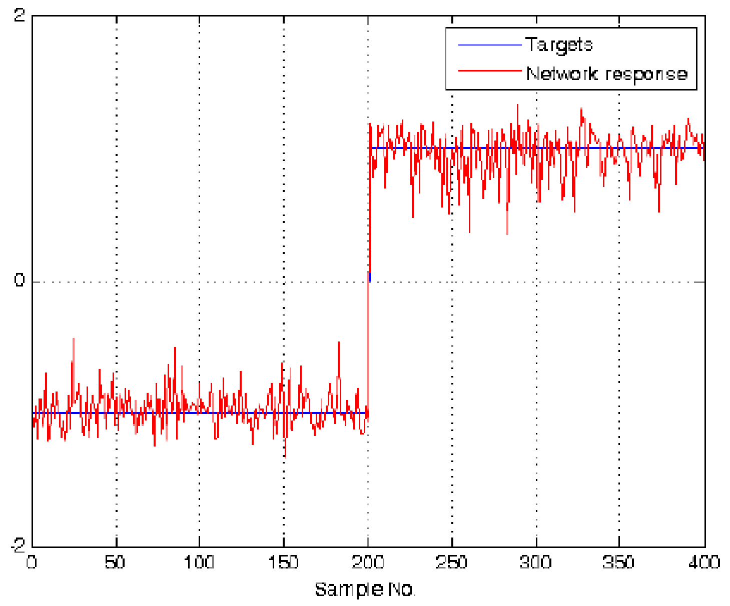
- Retrain a RBFN using Bayesian regularization backpropagation

```
% define custom training function: Bayesian regularization backpropagation
net.trainFcn='trainbr';
% perform Levenberg-Marquardt training with Bayesian regularization
net = train(net,P,T);
```

- Evaluate network performance after Bayesian regularization training

```
% check new RBFN spread
spread_after_training = net.b{1}
% simulate RBFN on training data
Y = net(P);
% calculate [%] of correct classifications
correct = 100 * length(find(T.*Y > 0)) / length(T);
fprintf('Num of neurons = %d\n',net.layers{1}.size)
fprintf('Correct class = %.2f %%\n',correct)
% plot targets and network response
figure;
plot(T')
ylim([-2 2])
set(gca,'ytick',[-2 0 2])
hold on
grid on
plot(Y','r')
legend('Targets','Network response')
xlabel('Sample No.')
```

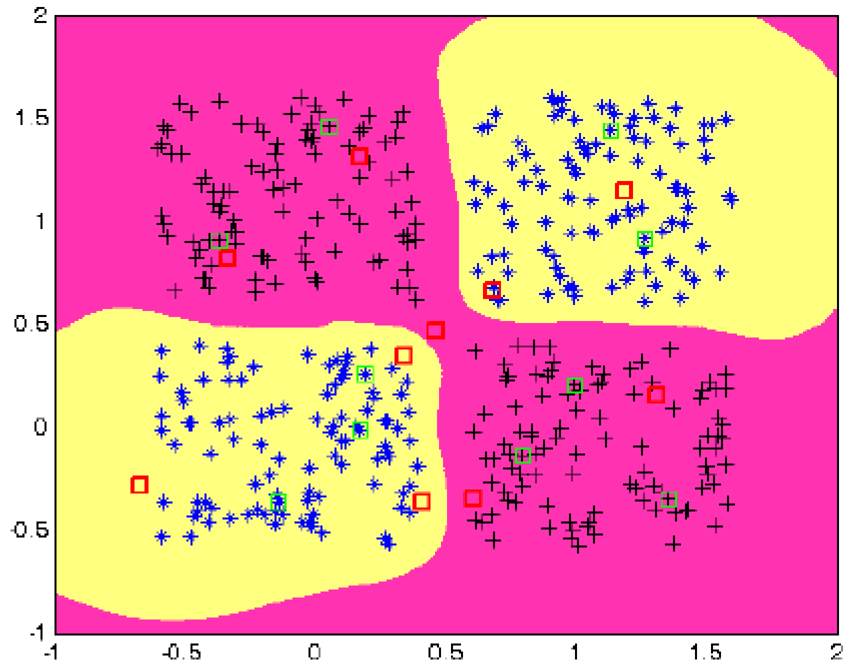
```
spread_after_training =
2.9924
3.0201
0.7809
0.5933
2.6968
2.8934
2.2121
2.9748
2.7584
3.5739
Num of neurons = 10
Correct class = 100.00 %
```



Bayesian regularization for RBFN (5/5)

- Plot classification result after Bayesian regularization training

```
% generate a grid
span = -1:.025:2;
[P1,P2] = meshgrid(span,span);
pp = [P1(:) P2(:)]';
% simualte neural network on a grid
aa = sim(net,pp);
% plot classification regions based on MAX activation
figure(1)
ma = mesh(P1,P2,reshape(-aa,length(span),length(span))-5);
mb = mesh(P1,P2,reshape( aa,length(span),length(span))-5);
set(ma,'facecolor',[1 0.2 .7],'linestyle','none');
set(mb,'facecolor',[1 1.0 .5],'linestyle','none');
view(2)

% plot modified RBFN centers
plot(net.iw{1}(:,1),net.iw{1}(:,2), 'rs', 'linewidth', 2);
```

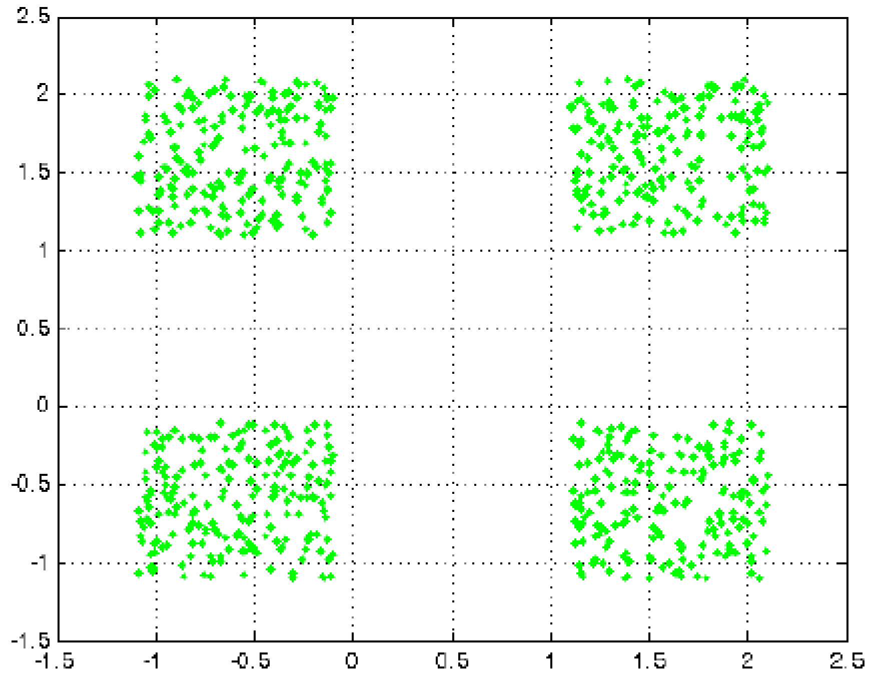


1D and 2D Self Organized Map

1D-SOM (1/2)

- Define 4 clusters of input data

```
close all, clear all, clc, format compact
% number of samples of each cluster
K = 200;
% offset of classes
q = 1.1;
% define 4 clusters of input data
P = [rand(1,K)-q rand(1,K)+q rand(1,K)+q rand(1,K)-q;
rand(1,K)+q rand(1,K)+q rand(1,K)-q rand(1,K)-q];
% plot clusters
plot(P(1,:),P(2,:),'g.')
hold on
grid on
```



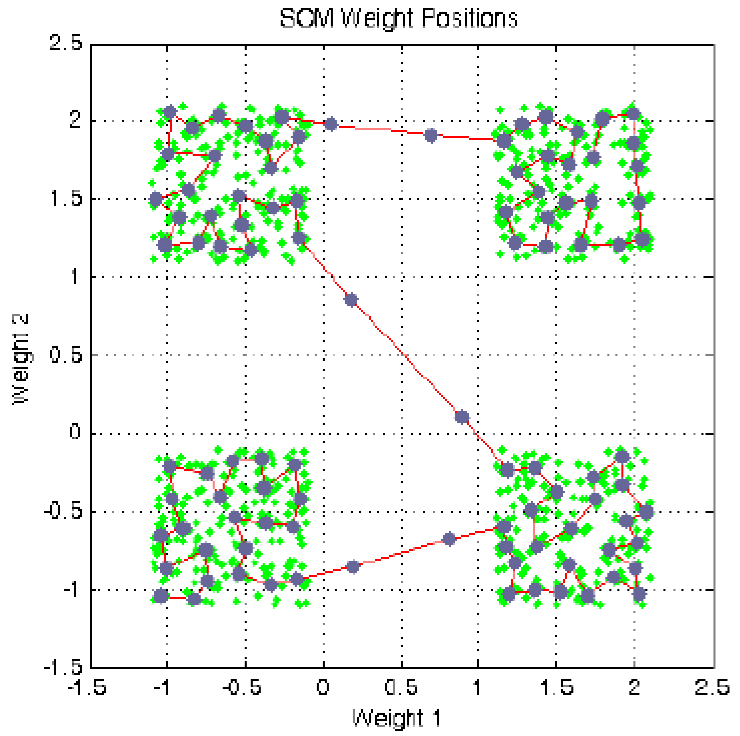
1D-SOM (2/2)

- Create and train 1D-SOM

```
% SOM parameters
dimensions = [100];
coverSteps = 100;
initNeighbor = 10;
topologyFcn = 'gridtop';
distanceFcn = 'linkdist';
% define net
net1 = selforgmap(dimensions,coverSteps, ...
   initNeighbor,topologyFcn,distanceFcn);
% train
[net1,Y] = train(net1,P);
```

- Plot 1D-SOM results

```
% plot input data and SOM weight positions
plotsompos(net1,P);
grid on
```



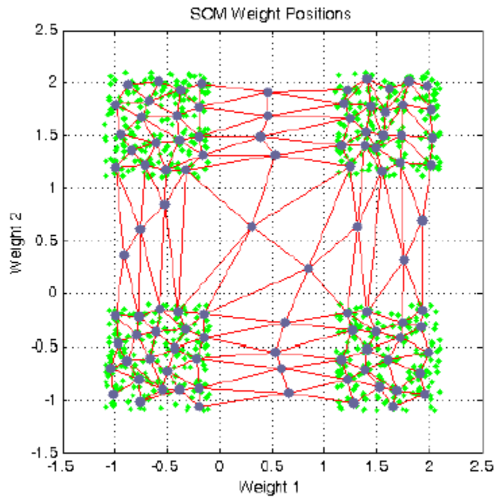
2D-SOM

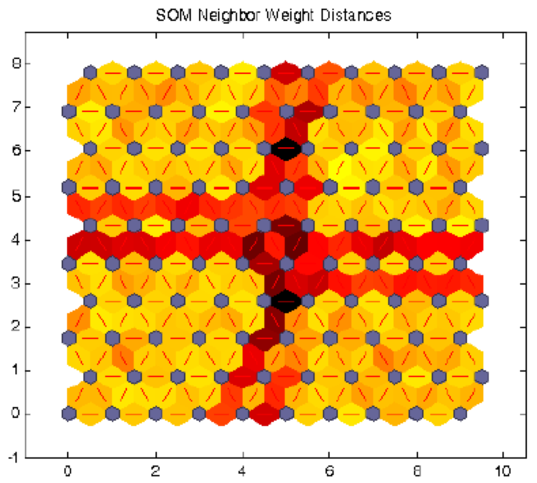
- Create and train 2D-SOM

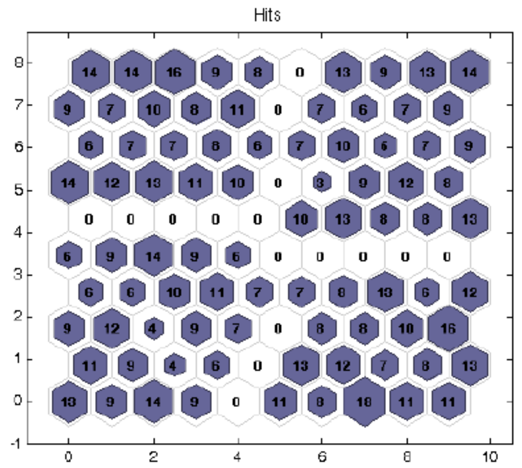
```
% SOM parameters
dimensions = [10 10];
coverSteps = 100;
initNeighbor = 4;
topologyFcn = 'hextop';
distanceFcn = 'linkdist';
% define net
net12 = selforgmap(dimensions,coverSteps, ...
   initNeighbor,topologyFcn,distanceFcn);
% train
[net2,Y] = train(net2,P);
```

- Plot 2D-SOM results

```
% plot input data and SOM weight positions
plotsompos(net2,P);
grid on
% plot SOM neighbor distances
plotsomnd(net2)
% plot for each SOM neuron the number of input vectors that it classifies
figure
plotsomhits(net2,P)
```







Exercises Part 7

1. Using One-Dimensional Self-organizing Map classify the following points into natural classes.
```
angles = 0:0.5*pi/99:0.5*pi;
X = [sin(angles); cos(angles)];
```

4. Using Two-Dimensional Self-organizing Map classify 1000 two-element vectors in a rectangle.
```
X = rands(2,1000);
```

Cross-Validation

Partition data for cross-validation (KFold)

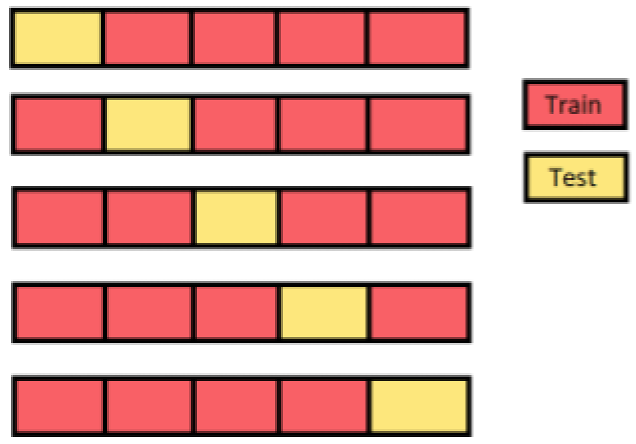
- *cvpartition* defines a random partition on a data set. Use this partition to define training and test sets for validating a statistical model using cross-validation. Use *training* to extract the training indices and *test* to extract the test indices for cross-validation. Use repartition to define a new random partition of the same type as a given *cvpartition* object.

```
c = cvpartition(n,'KFold',k)
c = cvpartition(group, 'KFold',k,'Stratify',stratifyOption)
```

- When you specify *group* as the first input argument, *cvpartition* discards rows of observations corresponding to missing values in group.

- If you specify 'Stratify',false, then *cvpartition* ignores the class information in group and creates a nonstratified random partition.
  Otherwise, the function implements stratification by default.

K-Fold for k=5

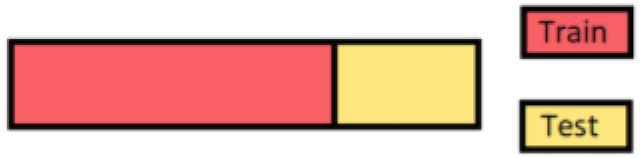


Partition data for cross-validation (Holdout)

- In the *non-exhaustive* method, we don't compute for all possible combinations of the original data. This is the simplest method of all. In Holdout validation, the data is randomly partitioned into train and test set.
  Most of the times it is 70/30 or 80/20 split.
  We train our model in the training set, and it'll be tested in the test set to see how well the model is performing for unknown events.

```
c = cvpartition(n,'Holdout',p)
c = cvpartition(group,'Holdout',p,'Stratify',stratifyOption)

idxTrain = training(c);
dataTrain = data(idxTrain, :);
idxTest = test(c);
dataTest = data(idxTest, :);
```

Holdout Validation- 70% train and 30% test



Partition data for cross-validation (Leaveout)

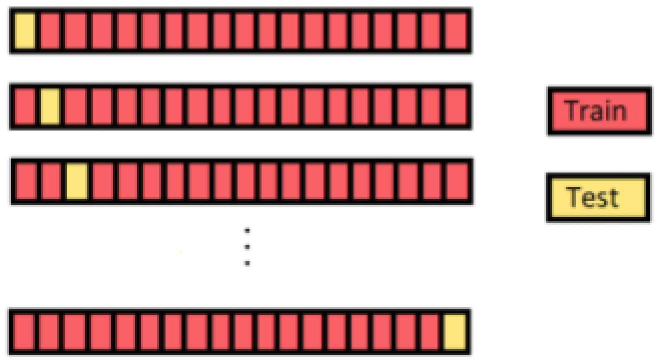
- *Leave-one-out* is a special case of *KFold* in which the number of folds equals the number of observations.
  In *LOO*, one observation is taken out of the training set as a validation set.
  We will train the model without this validation set and later test whether it correctly classify the observation.

c = cvpartition(n,'Leaveout')

c = cvpartition(n,'Resubstitution')

- *c* that does not partition the data.
  Both the training set and the test set contain all of the original n observations.

Leave-One-Out CV



Exercises Part 8

1. Train and test a pattern recognition neural network(Patternnet) using *dataset_classification.mat* performing *k-Fold* cross validation with *k = 10.*

2. Evaluate the performance by plotting the confusion matrix and ROC.
   Get the best network in terms of performance.