(Concepts of) Machine Learning

Shallow neural networks and learning rules

After this session you should be able to:

- Use fundamental neural network functions in Matlab;
- Implement simple weight updating in Matlab;
- Manipulate activation functions in Matlab;
- Describe what a decision boundary is and how it is used;
- Implement the perceptron rule in Matlab ;
- Experiment with the perceptron rule in Matlab.

What are Artificial Neural Networks (ANNs)?

They are massively parallel distributed processors made up of simple processing units called neurons, which have a natural propensity for storing experiential knowledge and making it available for use. Knowledge is acquired by the network from its environment through a learning process. This acquired knowledge is stored in the inter neuron connections or links known as synaptic weights.

The neuron as a computing element

A neuron is a simple information processing unit that is fundamental to the operation of a neural network. Fig. 1 shows the model of an artificial neuron.

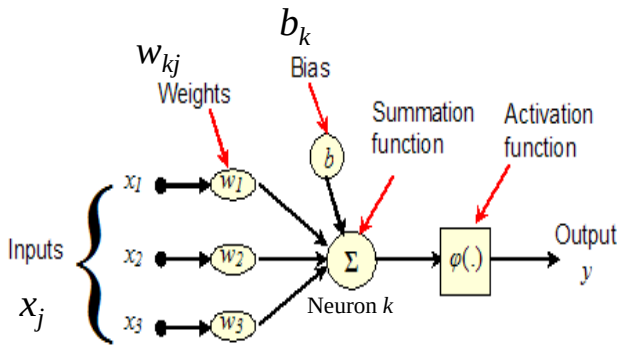


Figure 1: Nonlinear model of a neuron

There are four main elements in a neuron model, as shown in Fig. 1:

- **Synapses or Connecting links:** The neurons are connected by links, and each link has a numerical weight, or synaptic weight, associated with it. A signal x_j, at the input of the synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}. The first subscript refers to the neuron in question and the second subscript indicates the input end of the synapse to which the weight refers.
- **Summation Function or Linear Combiner:** Computes the weighted sum of all the input signals.
- **Activation or Transfer Function:** The neurons in the ANN transform their net input by using a scalar-to-scalar function called an activation function. These are needed to introduce non-linearity into the network and thus make it more powerful when performing classification tasks. There are many transfer functions available for use in the toolbox. The most commonly used ones are **`logsig`**, **`tansig`** and **`purelin`**.
- **Bias:** This has the effect of increasing or lowering, the net input of the activation function, depending on whether it is positive or negative.

Fundamentals of ANNs in Matlab

Some important concepts about ANN are described below:

- **Learning algorithm:** The procedure used to train a neural network to perform a particular task. The neurons are connected by links, and each link has a numerical weight associated with it. The weights express the strength or the importance of each neuron input. The neural network learns through repeated adjustments of these weights. Numerous learning functions are included in the toolbox such as **`learngdm`**, **`learnp`**, **`learnwh`** and others.
- **Network Architectures:** In a neural network, the neurons are organised in the form of layers. We consider two fundamentally different classes of network architectures:
  i. ***Single-Layer feed-forward or Static:*** This is the simplest form of layered network; there is an input layer of source nodes/neurons that projects onto an output layer of neurons, but not vice versa.
  ii. ***Multi-Layer feed-forward or Static:*** This category of networks has one or more hidden layers, whose computation nodes are referred to as the hidden units. The function of these hidden units is to intervene between the external input and the network output in some useful manner. These feed-forward networks are also referred to as static networks since there are no feedbacks or delays.

- **Data Structures:** The format of the data presented affects the networks training. There are two basic types of input vectors – ones that occur **concurrently** (at the same time or in no particular time sequence) and those that occur **sequentially** in time. For the concurrent vectors, the order in which the inputs are presented is not important, whereas for sequential vectors, the order in which these vectors appear is important.
- **Training Styles:** An ANN can be trained in 2 different ways. The first method is the **Incremental or online training** wherein the weights and biases of the network are updated each time an input is presented to the network. For this training mode the command, **adapt** is used. The other mode of training is the **Batch training** mode. Here the weights and biases are updated only after all the inputs have been presented. Batch training can be done using either **adapt** or **train**, although train is generally the best option.
- **Work Flow:** The work flow for the neural network design process has seven primary steps:
    (i) Collect data
    (ii) Create the network
    (iii) Configure the network
    (iv) Initialise the weights and biases
    (v) Train the network
    (vi) Validate the network
    (vii) Use the network

In the **Deep Learning Toolbox** (Matlab documentation available online at: https://uk.mathworks.com/help/deeplearning/index.html) a plethora of information about training shallow and deep networks as well as examples about each of the aforementioned functions are provided.

The case of a single neuron

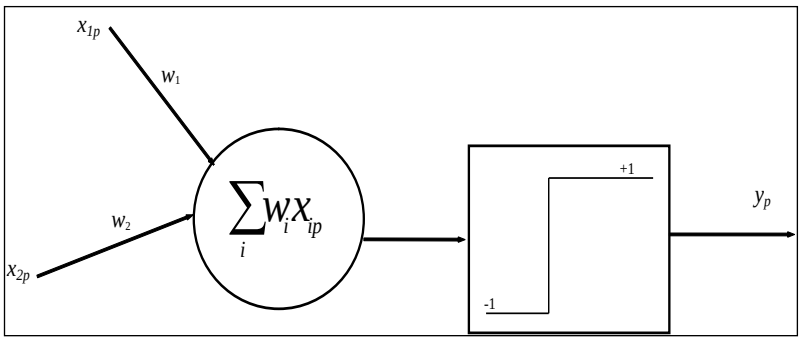
A single neuron with two weights is trained using the following rule:

$$w_i^{t+1} = w_i^t + \Delta w_i = w_i^t + \sum_{p=1}^{P} x_{ip}\, y_p, \;\; i = 1,2 \;,$$

where t denotes iterations, P is the number of patterns in the training set and the inputs, x_{ip} , and outputs, y_p , for each pattern, p=1,…,4, are shown in the following truth table

p	x_{1p}	x_{2p}	y_p

1	-1	-1	-1
2	-1	+1	-1
3	+1	-1	+1
4	+1	+1	-1



The weights are all zero at the start, i.e. $w_i^0 = 0, \text{for all } i = 1,2$. The node is trained by presenting it with input and output pairs in the same order as the truth table, starting form the top, then the first input pattern that is used (p=1) is [-1, -1] and the corresponding output pattern is (-1).

Matlab implementation

As the weights are initially set to zero, their value after a set of training patterns has been presented to the input is:

$$w_i^{t+1} = \sum_{p=1}^{P} x_{ip} y_p, \quad i = 1,2$$

This can be rewritten in matrix notation as:

$$[W]=[X]^T [Y]$$

where $[X]^T$ is the transpose of [X], implemented as – **`x'`** .

Define X and Y matrices in Matlab and calculate the weight matrix.
This can be simply done by defining the matrices X and Y and using Matlab operators for multiplication of matrices/vectors.

What is the neuron's output for the input data X when the hard limiter (threshold) activation function is used?
Use the **`hardlims`** function to calculate the neurons output. Does this meet your expectations?
Can you think of a way to improve the performance of the neuron?
Hint: Plot the output of the **`hardlims`** function when inputs are the interval [-5, 5] using steps of 0.1. Notice where the threshold is applied.

Perceptron Learning

Assume a training set P, shown in the table below, is presented at the input of the single neuron.

p	x_{1p}	x_{2p}	d_p
1	-0.5	-0.5	1
2	-0.5	+0.5	1
3	+0.3	-0.5	0
4	-0.1	+1	0
5	-0.8	0	0

A simple algorithm to train this model is perceptron learning which updates the weights according to the rule:

$$w^{t+1} = w^t + \varepsilon(d^t - y^t)x^t$$

where t denotes iterations, ε is the learning rate, P is the number of patterns in the training set; $d = \{d_p\}$ is the set of desired outputs (i.e. what we would like the neuron to produce), $x = \{x_{ip}\}$ is the input set, and $y = \{y_p\}$ is the set of real outputs produced (i.e. what the neuron really produces), where p=1,…,5, are shown in the above table.

The neuron operates by dividing the input space into two regions because it can only be in one of two states (i.e. 1 or 0 because a threshold activation is used). This kind of partitioning is called a decision boundary. In general, a decision boundary can take any functional form, but it is often useful to derive the optimal decision boundary that maximizes accuracy (best generalisation). In our case, the functional form is the equation of a line:

$$w_1x_1 + w_1x_2 = 0$$

This line defines the boundary between regions where the input pattern produces a positive response (output) and regions where the response will be zero.

Visit http://en.wikipedia.org/wiki/Decision_boundary for a relevant article.

Implementing perceptron leaning in Matlab

Let's take the following example of a four-class classification problem; the steps involved in perceptron learning are:

Step1: Defining data and classes
% number of samples of each class

```
K = 30;
```

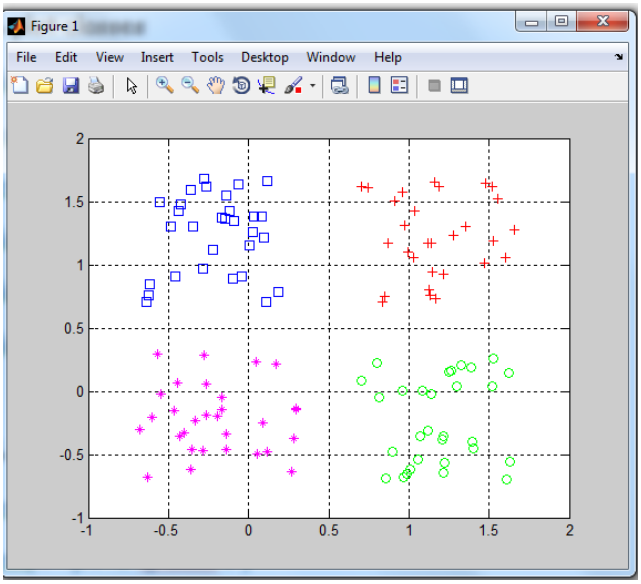
% define classes

```
q = .7;          % 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];
```

Step2: 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*')
```

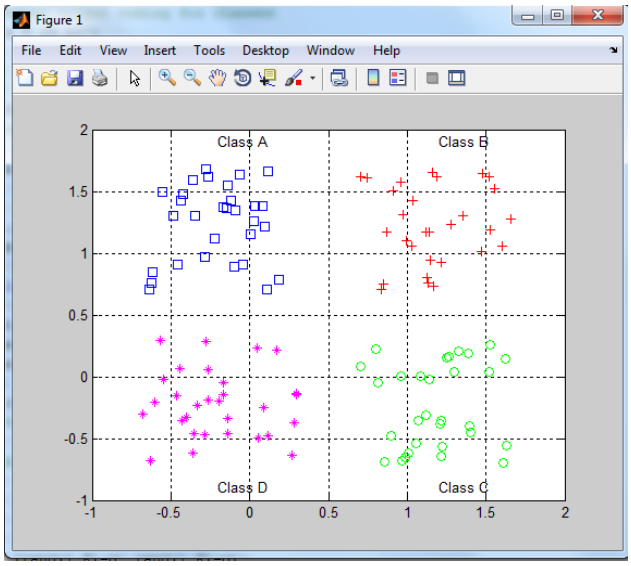
% the plot will look like this:



Step3: 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')
```

% plot should now look like this:



Step4: defining input & outputs
% define output coding for classes

```
a = [0 1]';
b = [1 1]';
c = [1 0]';
d = [0 0]';
```

% define inputs by combining inputs from all 4 classes

```
P = [A B C D];
```

% p → (2X120) size matrix containing all 4 classes; (:,1:30 → class A, (:,30:60 → class B & so on)

% define targets. Type T, below, in just one line – not two lines are shown below due to space limitations.

```
T = [repmat(a,1,length(A)) repmat(b,1,length(B))
 repmat(c,1,length(C)) repmat(d,1,length(D)) ];
```

% T → (2X120) size matrix containing desired targets for all 4 classes; [(:,1:30 → class A), (:,30:60 → class B & so on)]

Step5: network creation & training
% create a perceptron

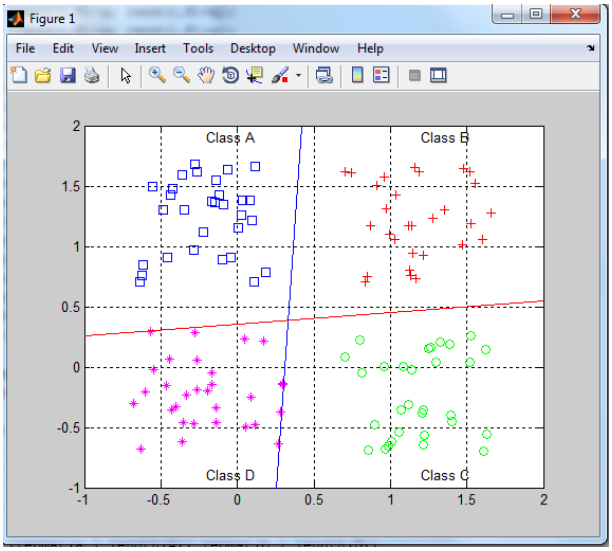
```
net = perceptron;
```

% note: you could also use the command `patternnet` instead, see `help patternnet`

% Perceptron training

```
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
```

% where, `net` → updated network; `Y` → network outputs and `E` → network error
% the plot should now look like:



Try changing class offset to 0.5 or 0.4, are the results different? Try increasing training passes? Does it help?

Does the perceptron classify the data correctly?

Retrain the perceptron again for 25 iterations. Is your result the same as before? Do you think it is possible to get better results?

Train the perceptron for 100 iterations. Are you able to obtain better classifications?