

Mustansiriyah University

College of Engineering

**Department of Computer Engineering and Artificial
Intelligence**

Artificial Intelligence Laboratory

For

Fourth Stage



Dr. Hussein Mohammed Ridha

4th Stage 1st semester	
Artificial Intelligence lab code: MSU0205010705 Hours/Units: 2/2	
1.	Simple Neuron and single Layer Feedforward Neural Network
2.	Multi-Layer Feedforward Neural Network
3.	Simulation with Concurrent and Sequential Inputs in Dynamic Networks
4.	Perceptron Learning Rules
5.	Design of Adaptive Linear Filter Using ADALINE
6.	Backpropagation Neural Networks
7.	Convolutional Neural Network (CNN)

Experiment No. (1)

Simple Neuron and single Layer Feedforward Neural Network

Introduction

Neuron consists of three basic components –weights, thresholds and a single activation function. An Artificial neural network (ANN) model based on the biological neural systems is shown in Figure 1.

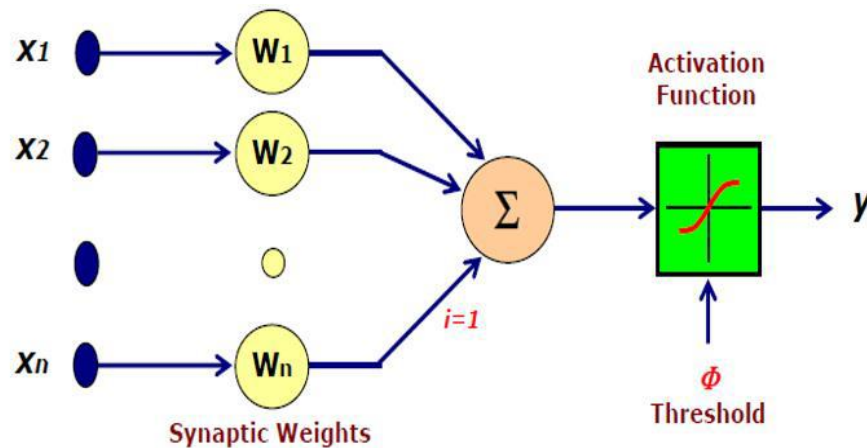


Figure 1. Basic Elements of Artificial Neural Network.

- **DIFFERENT LEARNING RULES**

A brief classification of Different Learning algorithms is depicted in figure 2.

- Training: It is the process in which the network is taught to change its weight and bias.
- Learning: It is the internal process of training where the artificial neural system learns to update/adapt the weights and biases.
- **Different Training /Learning procedures available in ANN are**
 - Supervised learning
 - Unsupervised learning
 - Reinforced learning
 - Hebbian learning
 - Gradient descent learning
 - Competitive learning
 - Stochastic learning

- **Requirements of Learning Laws**

- Learning Law should lead to convergence of weights
- Learning or training time should be less for capturing the information from the training pairs
- Learning should use the local information
- Learning process should be able to capture the complex nonlinear mapping available between the input & output pairs
- Learning should be able to capture as many patterns as possible
- Storage of pattern information's gathered at the time of learning should be high for the given network

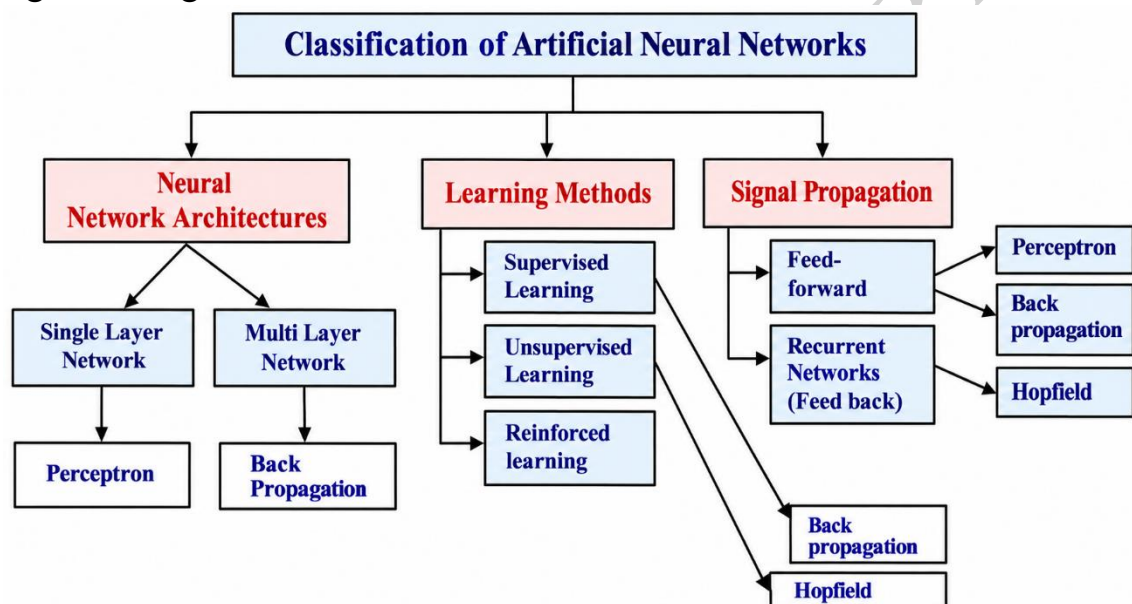


Figure 2. Different Training methods of Artificial Neural Network.

1. **Simple Neuron**

Object:

To design and test simple neurons with single scalar input and single vector input, using various transferred functions.

Theory:

A neuron with a single scalar input and no bias is shown in figure 3. The scalar input p is transmitted through a connection that multiplies its strength by the scalar weight w , to form the product wp , again a scalar.

Here the weighted input wp is the only argument of the transfer function f , which produces the scalar output a .

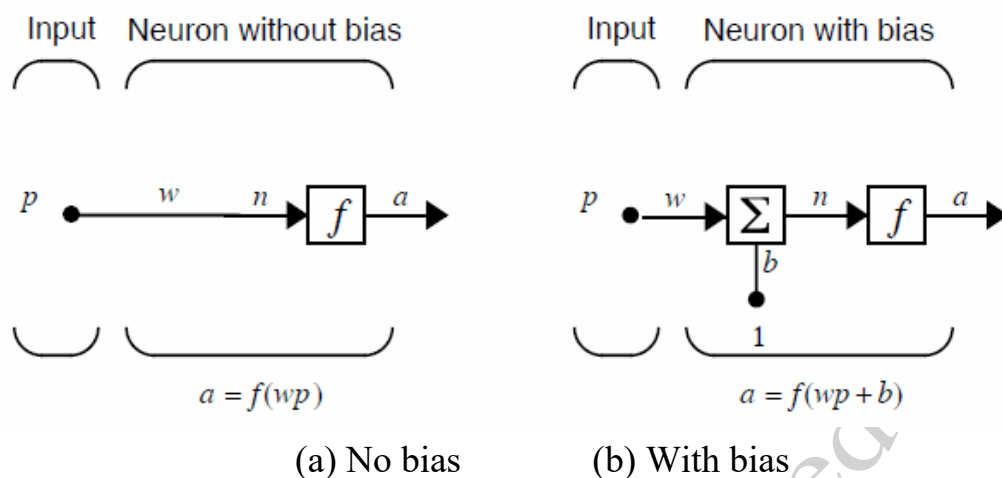


Figure 3. Simple Neuron.

The neuron shown in figure (1- b) has a scalar bias, b . The bias may be viewed as simply being added to the product wp as shown by the summing junction or as shifting the function f to the left by an amount b . The bias is much like a weight, except that it has a constant input of 1.

The transfer function net input n , again a scalar, is the sum of the weighted input wp and the bias b . This sum is the argument of the transfer function f which is typically a step function or a sigmoid function, which takes the argument n and produces the output a .

Note that w and b are both **adjustable** scalar parameters of the neuron.

The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior.

Transfer Functions

Many transfer functions can be used. A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve. The most used functions are shown below.

I. Threshold (hard limit) transfer function:

$$f(n) = \begin{cases} 0 & n < 0 \\ 1 & n \geq 0 \end{cases}$$

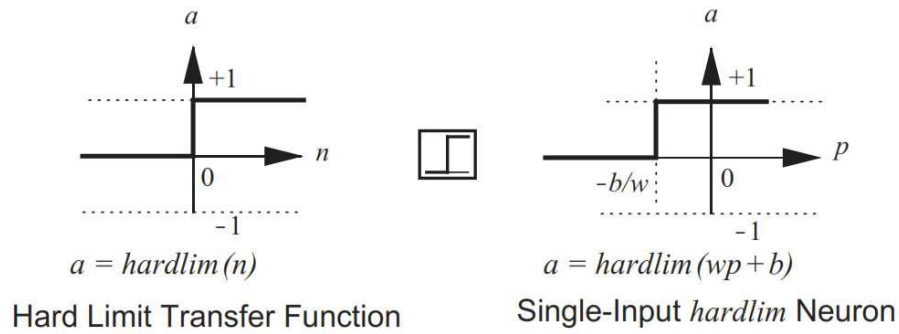


Figure 4. Hard Limit Transfer Function.

This function will be used to create neurons that classify inputs into two distinct categories.

II. Symmetric Hard Limit Transfer Function:

$$f(n) = \begin{cases} -1 & n < 0 \\ 1 & n \geq 0 \end{cases}$$

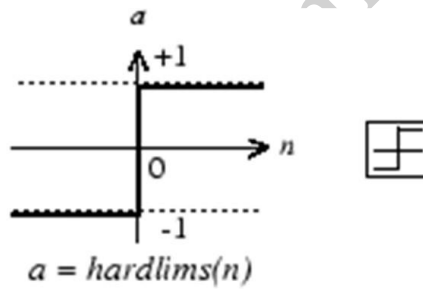


Figure 5. Symmetric Hard Limit Transfer Function.

III. Linear Transfer Function:

$$f(n) = n$$

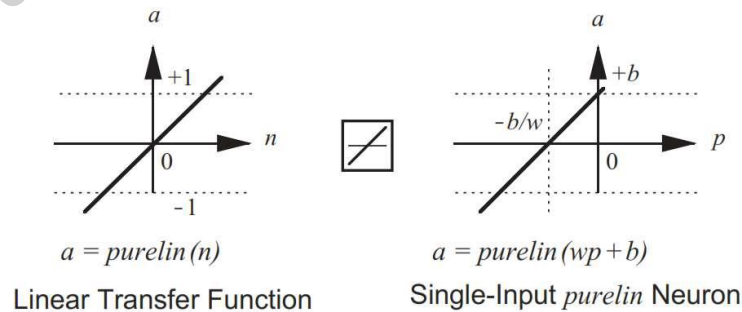


Figure 6. Linear Transfer Function.

IV. Logistic (Log-Sigmoid) Transfer Function:

Logistic function is a standard sigmoid function and is defined by

$$f(n) = \frac{1}{1 + e^{-n}}$$

The derivative of f is defined by $f'(n) = f(1 - f)$

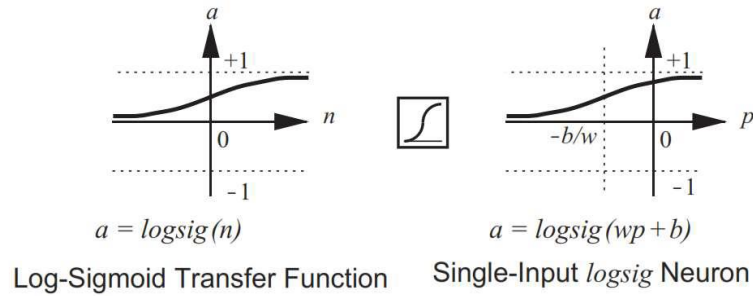


Figure 7. Log-Sigmoid Transfer Function.

The log-sigmoid transfer function is commonly used in multilayer networks that are trained using the backpropagation algorithm.

V. Hyperbolic Tangent Transfer Function:

The hyperbolic tangent is a sigmoid function and is defined by

$$f(n) = \frac{e^n - e^{-n}}{e^n + e^{-n}}$$

The derivative of f is defined by $f'(n) = (1 - f^2)$

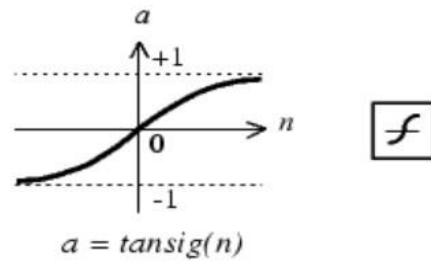
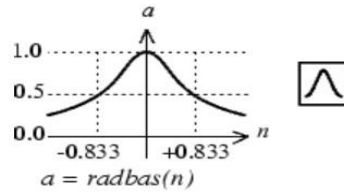


Figure 8. Hyperbolic Tangent Transfer Function.

Since $\frac{\tanh(\frac{n}{2})+1}{2} = \frac{1}{1+e^{-n}}$ then using the tanh function instead of the logistic one is equivalent. The tanh function has the advantage of being symmetrical with respect to the origin.

VI. Radial Basis Transfer Function:

$$f(n) = e^{-n^2}$$



Radial Basis Function

Figure 9. Radial Basis Transfer Function.

VII. Saturating Linear Transfer Function:

$$f(n) = \begin{cases} 0 & \text{if } n < 0 \\ n & \text{if } 0 \leq n \leq 1 \\ 1 & \text{if } n > 1 \end{cases}$$

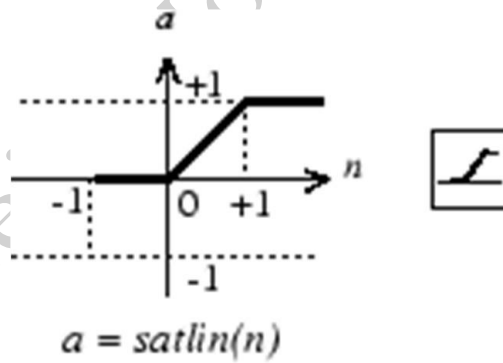


Figure 10. Saturating Transfer Function.

VIII. Symmetric Saturating Linear Transfer Function:

$$f(n) = \begin{cases} -1 & \text{if } n < -1 \\ n & \text{if } -1 \leq n \leq 1 \\ 1 & \text{if } n > 1 \end{cases}$$

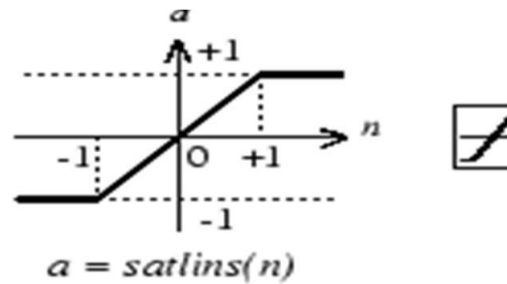


Figure 11. Symmetric Saturating Linear Transfer Function.

IX. Positive Linear Transfer Function:

$$f(n) = \begin{cases} 0 & \text{if } n < 0 \\ n & \text{if } n \geq 0 \end{cases}$$

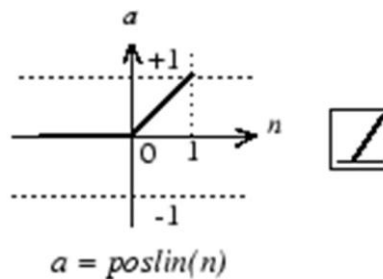


Figure 12. Positive Linear Transfer Function.

Neuron With Vector Input

A neuron with a single R -element input vector is shown below. Here the individual element inputs: $P_1, P_2, \dots, P_R$

are multiplied by weights: $w_{1,1}, w_{1,2}, \dots, w_{1,R}$

and the weighted values are fed to the summing junction. Their sum is simply $\mathbf{Wp}$, the dot product of the (single row) matrix $\mathbf{W}$ and the vector $\mathbf{p}$.

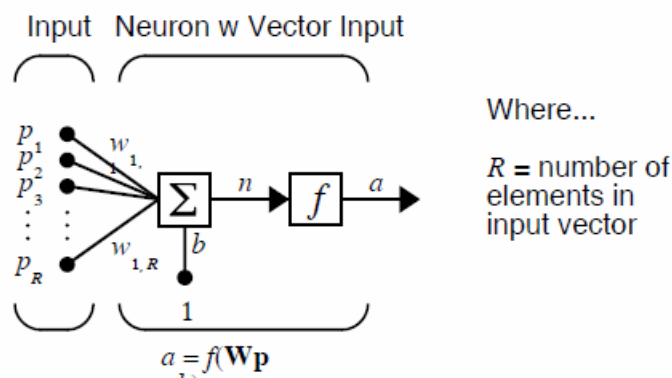


Figure 13. Neuron with input vector.

The neuron has a bias b , which is summed with the weighted inputs to form the net input n . This sum, n , is the argument of the transfer function f .

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$$

$$\mathbf{n} = \mathbf{W} * \mathbf{p} + \mathbf{b}$$

Abbreviated notations are illustrated in the diagram shown below.

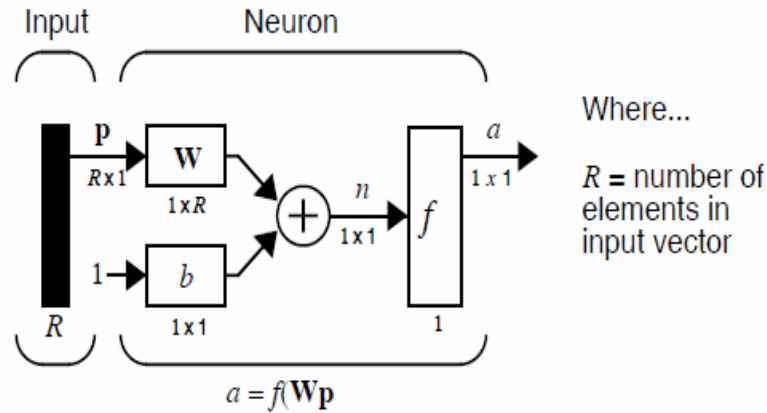


Figure 14. Single layer neural network.

Here the input vector $\mathbf{p}$ is represented by the solid dark vertical bar on the left. The dimensions of $\mathbf{p}$ are shown below the symbol $\mathbf{p}$ in the figure as $R \times 1$. Thus, $\mathbf{p}$ is a vector of R input elements. These inputs multiply the single row, R column matrix $\mathbf{W}$. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias b . The net input to the transfer function f is n , the sum of the bias b and the product $\mathbf{Wp}$. This sum is passed to the transfer function (f) to get the neuron's output a , which in this case is a scalar. Note that if we had more than one neuron, the network output would be a vector.

A *layer* of a network is defined in the figure shown above. A layer includes the combination of the weights, the multiplication and summing operation (here realized as a vector product $\mathbf{Wp}$), the bias b , and the transfer function f . The array of inputs, vector $\mathbf{p}$, is not included in or called a layer. Each time this abbreviated network notation is used, the size of the matrices will be shown just below their matrix variable names.

2. Single Layer Feed-forward Network

Objective: To design and test single feedforward neural networks utilizing various transfer functions with different patterns of inputs.

Theory:

In a layered neural network, the neurons are organized in the form of layers. In the simplest form of a layered network, we have an input layer of source nodes that project directly onto an output layer of neurons, (computation nodes), but not vice versa. In other words, this network of a feedforward type. **Fig. 14** illustrated R nodes in the input and S neurons in the output layers. Such a network is called a single-layer network. With the designation “single layer” referring to the output layer of computation nodes (neurons). We do not count the input layer of source nodes because no computation is performed there.

Each element in the source input node is either connected to each neuron in the output layer or some of them. Also, it is common for the number of inputs to a layer to be different from the number of neurons (i.e., $R \neq S$). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

In this network, each element of the input vector $\mathbf{p}$ is connected to each neuron input through the weight matrix $\mathbf{W}$. The i th neuron has a summer that gathers its weighted inputs and bias to form its own scalar output $n(i)$.

The various $n(i)$ taken together form an S -element net input vector $\mathbf{n}$. Finally, the neuron layer outputs from column vector $\mathbf{a}$. The expression for vector $\mathbf{a}$ is:

$$\mathbf{n} = f(\mathbf{W} * \mathbf{p} + \mathbf{b})$$

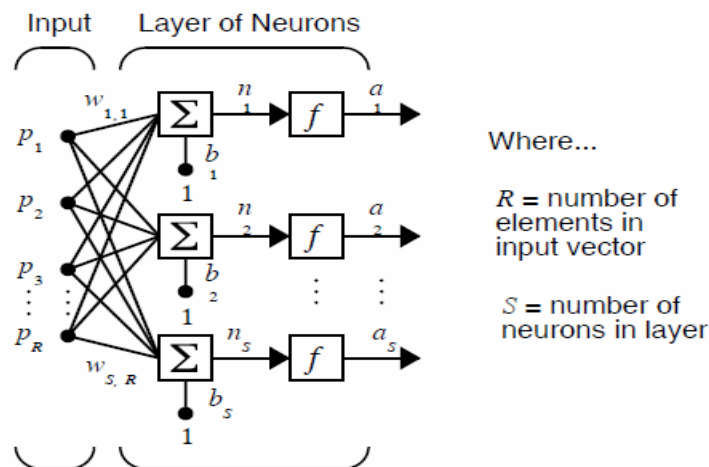


Figure 14. A one-layer network with R input elements and S neurons.

A single layer of neurons having different transfer functions can be created simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix $\mathbf{W}$.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix $\mathbf{W}$ indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices $w_{1,2}$ in say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The S neuron R input one-layer network also can be drawn in abbreviated notation as shown in figure 15.

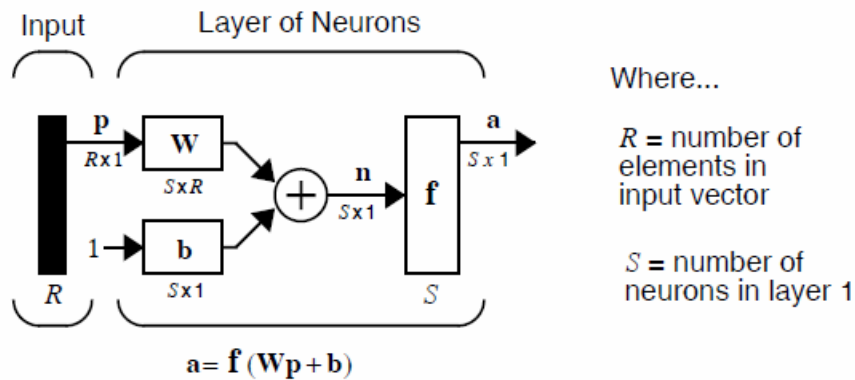


Fig 15. Abbreviated notation for the S neuron, R input single layer network.

Here $\mathbf{p}$ is an R length input vector, $\mathbf{W}$ is an $S \times R$ matrix, and $\mathbf{a}$ and $\mathbf{b}$ are S length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector $\mathbf{b}$, the summer, and the transfer function boxes.

Procedure:

Using MATLAB, write the appropriate code to test the output of a simple feedforward network having three input source node, two neurons in the output layer. Each source node is connected to each of the output neurons. The weight

matrix $W = \begin{bmatrix} 1 & 1 & 2 \\ -1 & -1.5 & 1 \end{bmatrix}$ with $b_1 = -0.5$, $b_2 = -2$. Two patterns of input vectors should be tested; $P_1^T = [1 \ -1 \ 1]$ and $P_2^T = [-1 \ 1 \ 1]$ using the following activation functions:

- A. Binary step function (hard-limit)
- B. Bipolar step function
- C. Sigmoid functions.

Dr. Hussein Mohammed Ridha



Results:

$$P_1^T = [1 \ -1 \ 1]$$

$$P_2^T = [-1 \ 1 \ 1]$$

	Binary	Sigmoid		Binary	Bipolar	Sigmoid
a	1	1		1	1	0.817
	0	-1		0	-1	0.183

Experiment No. (2)

Multi-Layer Feedforward Neural Network

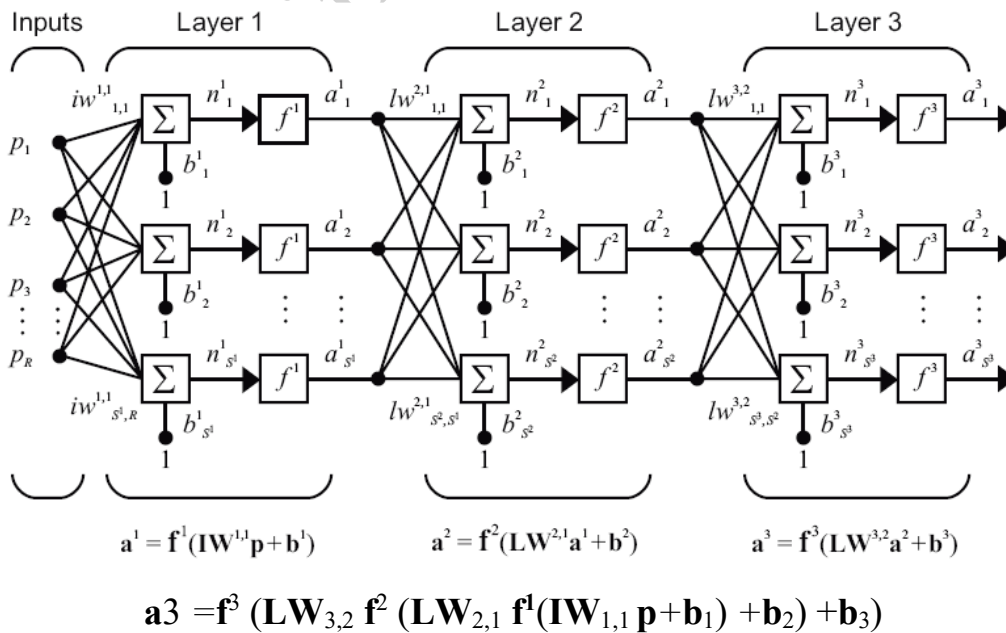
Object:

Design and test Multi-layer feedforward neural network with different pattern inputs using various transferred functions.

Theory:

A multi-layer feedforward neural network distinguishes itself by the presence of one or more hidden layers. The source nodes in the input layer of the network constitute the input signals applied to the neurons in the 2nd. layer (i.e., the first hidden layer). The output signals of the second layer are used as input to the 3rd. layer, and so on for the rest of the network. This approach can be taken with any layer of the network. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*.

A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in our figures, we append the number of the layer as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown below, and in the equations at the bottom of the figure.



This network has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different neurons. A constant input 1 is fed to the biases for each neuron.

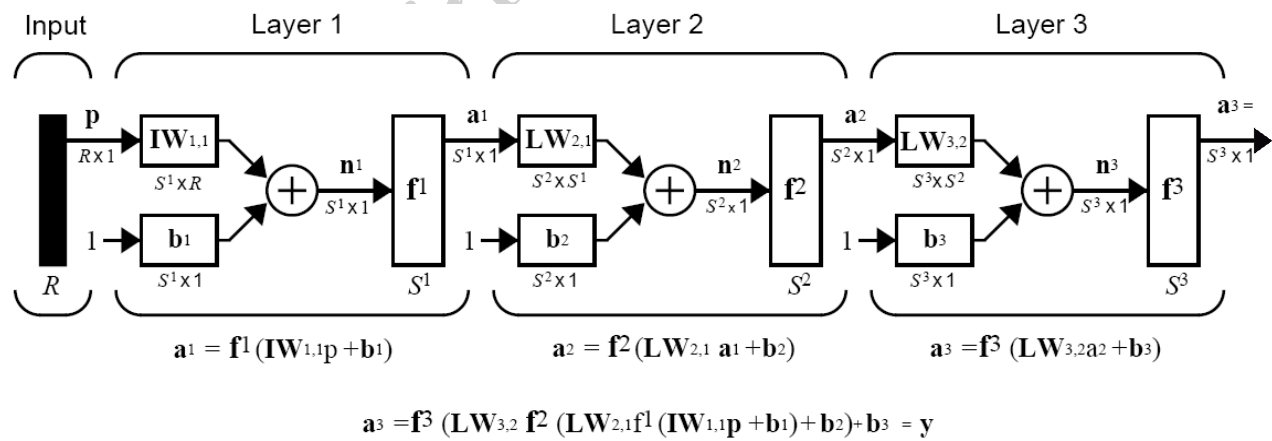
As mentioned earlier, in a multi-layer network, the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S^1 inputs, S^2 neurons, and an $S^2 \times S^1$ weight matrix W^2 . The input to layer 2 is $\mathbf{a}^1$; the output is $\mathbf{a}^2$. Now that we have identified all the vectors and matrices of layer 2, we can treat it as a single-layer network on its own.

The three-layer network shown in the figure above has one output layer (layer 3) and two hidden layers (layer 1 and layer 2).

The same three-layer network discussed previously also can be drawn using our abbreviated notation.

Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in “Backpropagation”.

Here it is assumed that the output of the third layer, $\mathbf{a}^3$, is the network output of interest, and we have labeled this output as $\mathbf{y}$. We will use this notation to specify the output of multilayer networks.



Procedure:

1- Write the appropriate code to calculate the outputs of the two-layer feedforward network shown in the figure below, which have the following:

There is a set of three input vectors, each with two elements:

$$p_1^T = [1 \ 2], p_2^T = [-1 \ 1], p_3^T = [1 \ -2]$$

There are **four** neurons in the first (hidden) layer and **three** neurons in the second (output) layer. The four hidden neurons are totally linked with the three output neurons. The transfer function in the first layer is tan-sigmoid (tanh), and the output layer transfer function is linear.

The bias vectors are:

$$b_1^T = [-.5 \ 1 \ 0 \ -0.5], b_2^T = [1 \ 1 \ -.5]$$

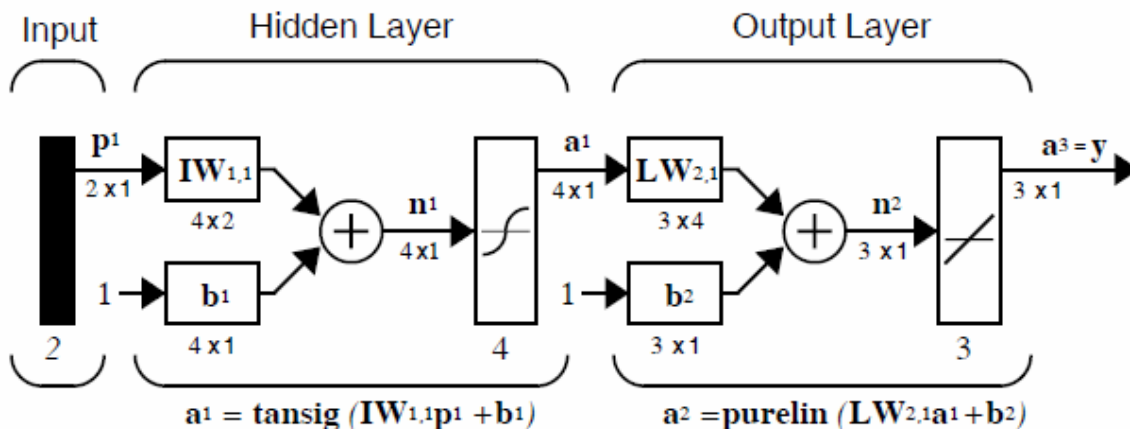
The weight matrices are:

$$IW_{1,1} = [1 \ .5; 1 \ -.5; 1.5 \ .5; 1 \ -1]$$

$$LW_{2,1} = [1 \ 1 \ .3 \ -.2; .4 \ .5 \ -.5 \ 1.5; .3 \ 1 \ 1 \ .6]$$

2- Replace the transfer function in the first layer to be log-sigmoid and calculate the outputs.

3- As in (2) but with transfer function of hardlims (bipolar step function), and $b_2^T = [-.5 \ .6 \ 1]$.



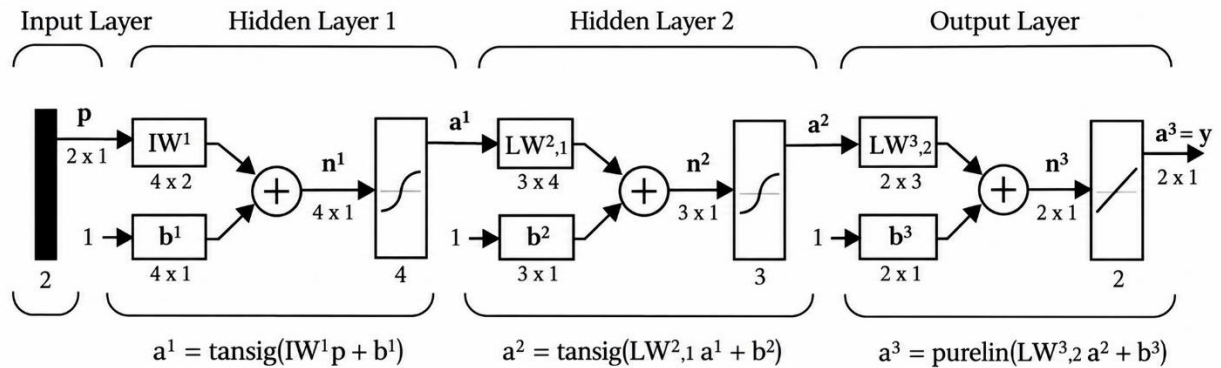
Results:

tanh	IW		p	b1	n1	a1	LW2				b2	n2	a2
	1	0.5	1	-0.5	1.5	0.91	1	1	0.3	-0.2	1	3.14	3.14
	1	-0.5	2	1	1	0.76	0.4	0.5	-0.5	1.5	1	-0.11	-0.11
	1.5	0.5		0	2.5	0.99	0.3	1	1	0.6	-0.5	0.98	0.98
	1	-1		-0.5	-1.5	-0.91							
logsig	1	0.5	1	-0.5	1.5	0.82	1	1	0.3	-0.2	1	2.79	2.79
	1	-0.5	2	1	1	0.73	0.4	0.5	-0.5	1.5	1	1.5	1.5
	1.5	0.5		0	2.5	0.92	0.3	1	1	0.6	-0.5	1.51	1.51
	1	-1		-0.5	-1.5	0.18							
hardlim s	1	0.5	1	-0.5	1.5	1	1	1	0.3	-0.2	1	2	2
	1	-0.5	2	1	1	1	0.4	0.5	-0.5	1.5	1	-0.5	-0.5
	1.5	0.5		0	2.5	1	0.3	1	1	0.6	-0.5	2.7	2.7
	1	-1		-0.5	-1.5	-1							

tanh	IW		p	b1	n1	a1	LW2				b2	n2	a2
	1	0.5	-1	-0.5	-1	-0.76	1	1	0.3	-0.2	1	-0.25	-0.25
	1	-0.5	1	1	-0.5	-0.46	0.4	0.5	-0.5	1.5	1	-0.63	-0.63
	1.5	0.5		0	-1	-0.76	0.3	1	1	0.6	-0.5	-2.54	-2.54
	1	-1		-0.5	-2.5	-0.99							
logsig	1	0.5	-1	-0.5	-1	0.27	1	1	0.3	-0.2	1	1.71	1.71
	1	-0.5	1	1	-0.5	0.38	0.4	0.5	-0.5	1.5	1	1.28	1.28
	1.5	0.5		0	-1	0.27	0.3	1	1	0.6	-0.5	0.27	0.27
	1	-1		-0.5	-2.5	0.08							
hardlim s	1	0.5	-1	-0.5	-1	1	1	1	0.3	-0.2	1	2	2
	1	-0.5	1	1	-0.5	1	0.4	0.5	-0.5	1.5	1	-0.5	-0.5
	1.5	0.5		0	-1	1	0.3	1	1	0.6	-0.5	2.7	2.7
	1	-1		-0.5	-2.5	-1							

tanh	IW		p	b1	n1	a1	LW2				b2	n2	a2
	1	0.5	1	-0.5	-0.5	-0.46	1	1	0.3	-0.2	1	1.47	1.47
	1	-0.5	-2	1	3	1	0.4	0.5	-0.5	1.5	1	2.56	2.56
	1.5	0.5		0	0.5	0.46	0.3	1	1	0.6	-0.5	1.41	1.41
	1	-1		-0.5	2.5	0.99							
logsig	1	0.5	1	-0.5	-0.5	0.38	1	1	0.3	-0.2	1	2.33	2.33
	1	-0.5	-2	1	3	0.95	0.4	0.5	-0.5	1.5	1	2.7	2.7
	1.5	0.5		0	0.5	0.62	0.3	1	1	0.6	-0.5	1.74	1.74
	1	-1		-0.5	2.5	0.92							
hardlim s	1	0.5	1	-0.5	-0.5	-1	1	1	0.3	-0.2	1	-0.5	-0.5
	1	-0.5	-2	1	3	1	0.4	0.5	-0.5	1.5	1	1.7	1.7
	1.5	0.5		0	0.5	1	0.3	1	1	0.6	-0.5	3.3	3.3
	1	-1		-0.5	2.5	1							

Exercise:



```

clc;
clear all;
close all;
% Input vectors
P1 = [1 2]';
P2 = [-1 1]';
P3 = [1 -2]';
% Weights for first hidden layer (4x2)
W1 = [1 0.5;
      1 -0.5;
      1.5 0.5;
      1 -1];
% Weights for second hidden layer (3x4)
W2 = [1 1 0.3 -0.2;
      0.4 0.5 -0.5 1.5;
      0.3 1 1 1.6];
% Weights for output layer (2x3)
W3 = [0.5 1 -0.5;
      1 -0.5 0.8];
% Bias vectors
b1 = [-0.5 1 0 -0.5]';
b2 = [1 1 -0.5]';
b3 = [0.5 -1]';
% Calculate outputs for each input
P = [P1 P2 P3];

% First hidden layer
n1 = W1*P + b1
a1 = tansig(n1)
% Second hidden layer
n2 = W2*a1 + b2
a2 = logsig(n2)
% Third layer (output)
n3 = W3*a2 + b3
a3 = purelin(n3)

```

Experiment No. (3)

Simulation with Concurrent and Sequential Inputs in Dynamic Networks

Object:

The objective of this experiment is to simulate static inputs and sequential dynamic inputs applied to a dynamic neural network, and then to show how the format of input data structures affects the simulation of networks.

Theory:

In dynamic system feedback is said to exist whenever the output of an element in the system influences in part the input applied to that element, thereby giving rise to one or more closed paths for the transmission of signals around the system. For example, such system may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons, as illustrated in figure 1. In this figure, there are no self-feedback loops in the network (self-feedback refers to a situation where the output of a neuron is fed back into its own input).

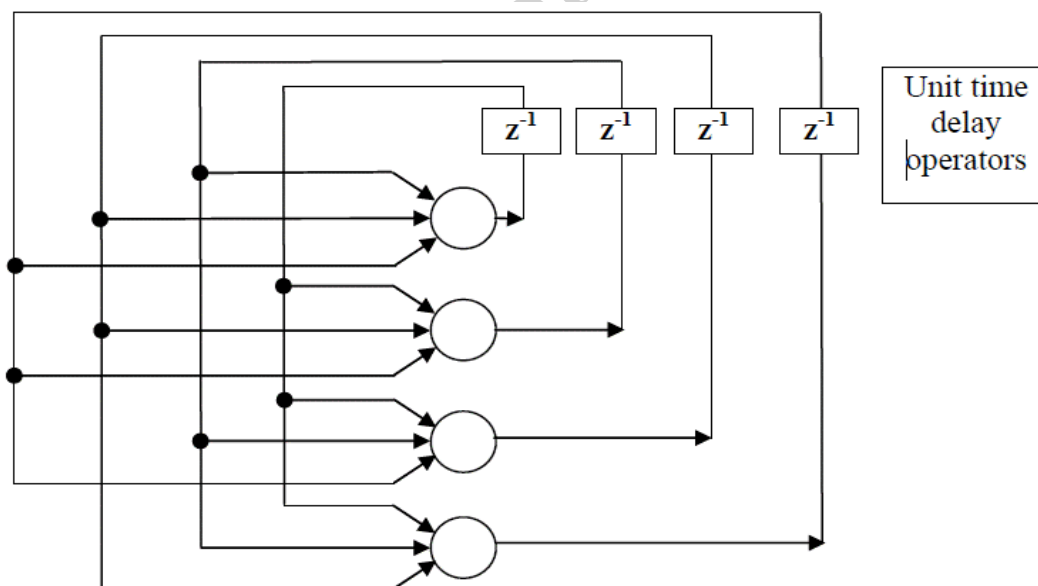


Figure 1. Single layer feedback network with no self-feedback loops.

The network illustrated in this figure also has no hidden neurons. Figure (2) illustrates another class of dynamic networks with hidden neurons. The feedback connections in this fig. originate from the hidden neurons as well as from the output neurons.

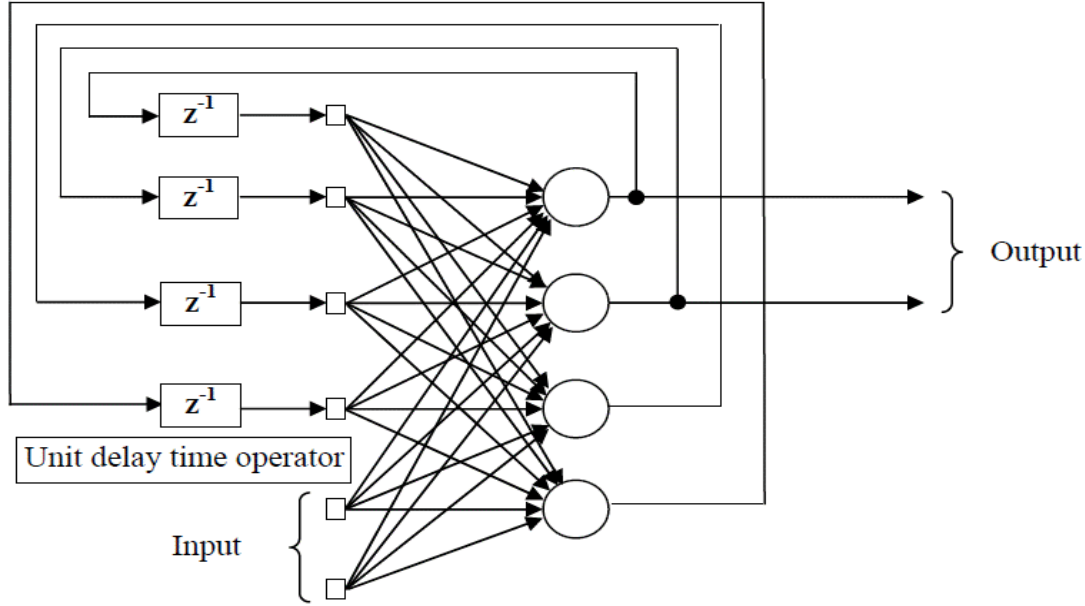


Figure 2. Another class of dynamic system.

The input signal $p(t)$ in such systems are functions of the discrete – time variable (t). And $p(t-1)$ is a sample of the input signal delayed by **1-time** units. Accordingly, we may express the output signal $a_k(t)$ as an infinite weighted summation of present and past samples of the input signal $p_j(t)$, as shown by:

$$a_k(t) = \sum_{l=0}^{\infty} w^{l+1} p_j(t-l)$$

We now see clearly that the dynamic behavior of a feedback system is controlled by the weight w .

In particular two specific cases can be distinguished:

1- $|w| < 1$, the output signal $a_k(t)$ is exponentially convergent, that is the **system is stable**. This case is illustrated in fig. 3-a for a positive w .

2- $|w| \geq 1$, the output signal $a_k(t)$ is divergent, that is, the **system is unstable**.

If $|w| = 1$ the divergence is linear, as in fig. 3-b,

and if $|w| > 1$ the divergence is exponential, as in fig. 3-c.

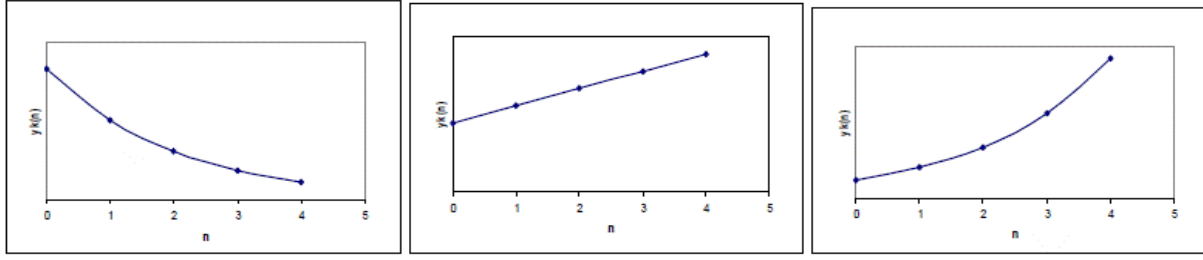


Figure 3. The response of different values of w .

Now, suppose that, for some power N , $|w|$ is small enough relative to the unity such that w^N is negligible for all practical purposes. In such situation we may approximate the output a_k by finite sum

$$\begin{aligned} a_k(t) &\approx \sum_{l=0}^{N-1} w^{l+1} p_j(t-l) \\ &= wp_j(t) + w^2 p_j(t-1) + w^3 p_j(t-2) + \dots + w^N p_j(t-N+1) \end{aligned}$$

Simulation with Static and dynamic inputs

We are concerned with two basic types of input vectors: those that occur **concurrently** (at the same time, or in no particular time sequence), and those that occur **sequentially** in time. For **concurrent vectors, the order is not important**, and if we had a number of networks running in parallel, we could present one input vector to each of the networks.

For **sequential vectors, the order in which the vectors appear is important**.

Simulation With Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, we do not have to be concerned about whether or not the input vectors occur in a particular time sequence, so we can treat the inputs as concurrent. In addition, we make the problem even simpler by assuming that the network shown in fig. (4) has only one input vector.

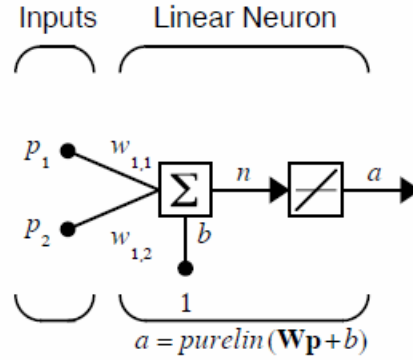


Figure 4. A static network.

Simulation With Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this, we use a simple network that contains one delay as shown in fig. (5).

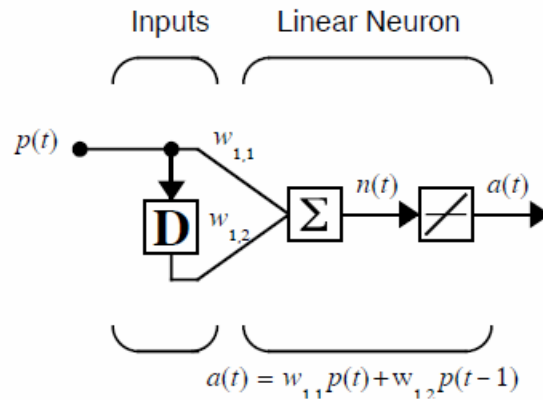


Figure 5. A dynamic network.

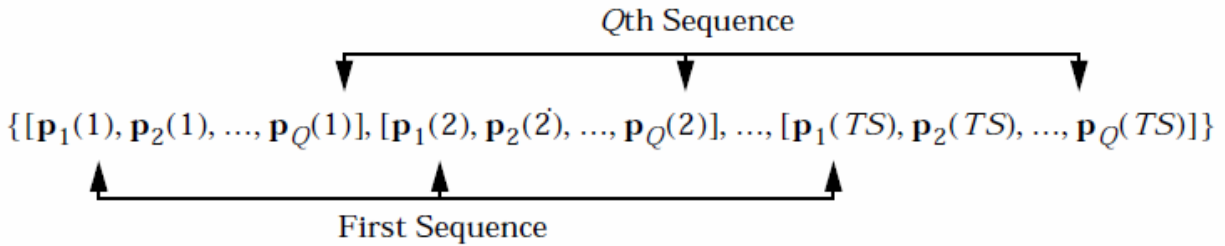
Simulation With Concurrent Inputs in a Dynamic Network

If we were to *apply the same inputs* from the previous section as a *set of concurrent inputs instead of a sequence of inputs*, we would obtain a *completely different response*. It would be as if *each input were applied concurrently to a separate parallel network and computed one output*.

Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero.

In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences.

The following diagram shows the general format for the input P to the `sim` function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.



Procedure:

In this experiment, we shall apply concurrent inputs to static networks. Then we shall apply sequential and concurrent inputs to dynamic networks. Note that it is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained.

We will begin with static networks and then move to dynamic networks.

1- Use the network shown in figure (4) as an example of a static network.

A- Write the appropriate code to simulate the network. Assign the weight matrix and bias to $\mathbf{W} = [1 \ 2]$, $\mathbf{b} = [0]$. Suppose that the network simulation data set consists of $q = 4$ concurrent vectors:

$$p_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad p_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad p_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad p_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix},$$

B- To set up this feedforward network, try simulating this network again using the following commands and compare the results:

→ create a new network layer

net = newlin([1 3;1 3],1);

where, **newlin(PR, S, ID, LR)** takes these arguments,

PR - $R \times 2$ matrix of min and max values for R input elements.

S - Number of elements in the output vector.

ID - Input delay vector, default = [0].

LR - Learning rate, default = 0.01.

and returns to a new linear layer.

net = newlin(PR, S, 0, P) takes an alternate argument,

P - Matrix of input vectors.

and returns to a linear layer with the maximum stable learning rate for learning with inputs **P**.

→ Assign the weight matrix and bias

net.IW{1,1} = [1 2];

net.b{1} = 0;

where, IW referred to the input layer's weights.

→ Concurrent vectors are presented to the network as a single matrix:

P = [1 2 2 3; 2 1 3 1];

→ Simulate the network using the following command:

A = sim(net,P)

Where, the 'sim' function simulates neural networks using the input **p**.

2- Use the network shown in fig. (5) as an example of a dynamic network that contains one delay.

A- Write the appropriate code to simulate the network. Assign the weight matrix and bias to **W** = [1 2], **b** = [0]. Suppose that the input sequence is:

p1 = [1], **p2** = [2], **p3** = [3], **p4** = [4].

B- Try simulating this network again using the following commands and compare the results:

→ Create the network:

net = newlin([-1 1], 1, [0 1]);

net.biasConnect = 0;

where, **biasConnect** property defines which layers have biases. **net.biasConnect** can be set to any N_I -by-1 matrix of Boolean values, where N_I is the number of network layers (net.numLayers). The presence (or absence) of a bias to the i th layer is indicated by a 1 (or 0) at: **net.biasConnect(i)**

Side Effects. Any change to this property alters the presence or absence of structures in the cell array of biases (**net.biases**) and in the presence or absence of vectors in the cell array of biases (**net.b**).

→ Assign the weight matrix:

net.IW{1,1} = [1 2];

→ Sequential inputs are presented as elements of a cell array:

P = {1 2 3 4};

→ Simulate the network:

A = sim(net,P)

As you can see, we input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result.

3- Use the network shown in fig. (5) as an example of a dynamic network that can be simulated with a concurrent set of sequences.

we wanted to present the following two sequences to the network:

p₁(1) = 1, p₁(2) = 2, p₁(3) = 3, p₁(4) = 4

p₂(1) = 4, p₂(2) = 3, p₂(3) = 2, p₂(4) = 1

A- Write the appropriate code to simulate the network. Assign the weight matrix and bias to be as mentioned in (1-a) above.

B- Try simulating this network again using the following commands and compare the results:

→ The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

P = {[1 4] [2 3] [3 2] [4 1]};

→ Simulate the network:

A = sim(net,P);

A_{ij}

We can see from the results that the first column of each matrix makes up the output sequence produced by the first input sequence. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

Dr. Hussein Mohammed Ridha

Procedure results:

1. Simulation with Concurrent Inputs in a Static Network

% a

P=[1 2 2 3;2 1 3 1];

W=[1 2]; B=[0];

A=purelin(W*P+B)

% b

net=newlin([1 3], [1 3],1);

net = linearlayer;

net.inputs{1}.size = 2;

net.layers{1}.dimensions = 1;

net.IW{1,1}=[1 2];

net.b{1}=0;

A=sim(net,P)

A = 5 4 8 5

2. Simulation with Sequential Inputs in a Dynamic Network

p=[5 3 4 5 8];

w=[2 5]; b=[0];

prev=0;

for t =1:length(p)

a(t)=w(1)*p(t)+w(2)*prev;

prev=p(t);

end

A=purelin(a)

%% Checking

net = linearlayer([0 1]);

net.inputs{1}.size = 1;

net.layers{1}.dimensions = 1;

net.biasConnect = 0;

% net.layers{1}.transferfcn='logsig';

net.IW{1,1} = [2 5];

P = {5 3 4 5 8};

A = net(P)

3. Simulation with concurrent set of sequences in a Dynamic Network

```
P={ [1 4] [2 3] [3 2] [4 1] };  
net=linearlayer([0 1]);  
net.inputs{1}.size=1;  
net.layers{1}.dimensions=1;  
net.biasConnect=0;  
net.IW{1,1}=[1 2];  
% net.layers{1}.transferfcn='purelin';  
A=sim(net,P)  
A{:}
```

The result output of part 3 would be $A = \{ [1 \ 4] \ [4 \ 11] \ [7 \ 8] \ [10 \ 5] \}$

Experiment No. (4) Perceptron Learning Rules

Object:

This experiment has several objectives, in which we introduce Perceptron learning rules, methods of deriving the next changes that might be made in a network, and training, a procedure whereby a network is adjusted to do a particular job. The experiment also covers functions to initialize and simulate such networks.

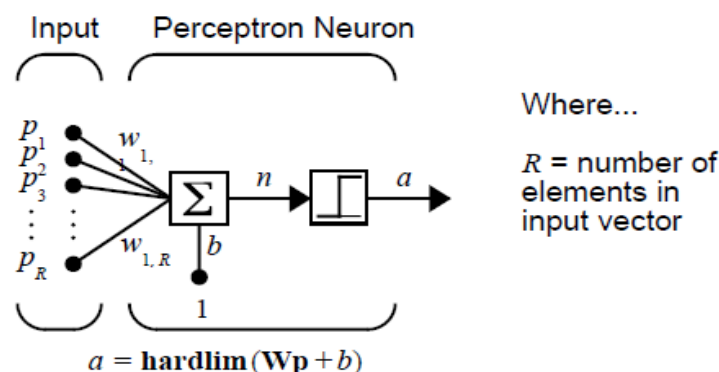
Theory:

Rosenblatt [Rose61] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections.

The main feature of their neuron model is that a weighted sum of input signals is compared to a threshold to determine the neuron output. They went on to show that networks of these neurons could, in principle, compute any arithmetic or logical function. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

Neuron Model

A perceptron neuron, which uses the **hard-limit transfer function (hardlims)**, is shown below. Each external input is weighted with an appropriate weight w_{1j} , and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The returned values from hard-limit transfer function are either a 0 or a 1.

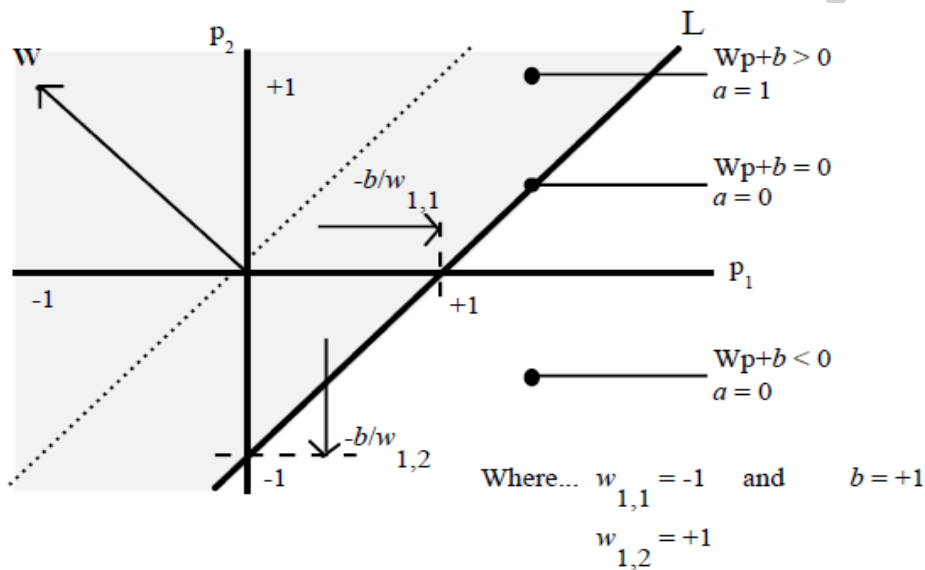


The perceptron neuron produces a 1 if the net input into the transfer function is greater than 0; otherwise, it produces a 0 (in MATLAB, the perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise, it produces a 0).

The network output is given below:

$$a = \text{hardlim}(\mathbf{Wp} + b)$$

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. The input space of a two-input hard limit neuron with the weights $w_{1,1} = -1$, $w_{1,2} = 1$ and a bias $b = 1$, is shown below. Two classification regions are formed by the *decision boundary* line L at $\mathbf{Wp} + b = 0$.



This line is perpendicular to the weight matrix $\mathbf{W}$ and shifted according to the bias b . Input vectors above and to the left of line L will result in a net input greater than 0; and therefore, cause the hard-limit neuron to output a 1.

Input vectors below and to the right of line L cause the neuron to output 0. The dividing line can be oriented and moved anywhere to classify the input space as desired by picking the weight and bias values.

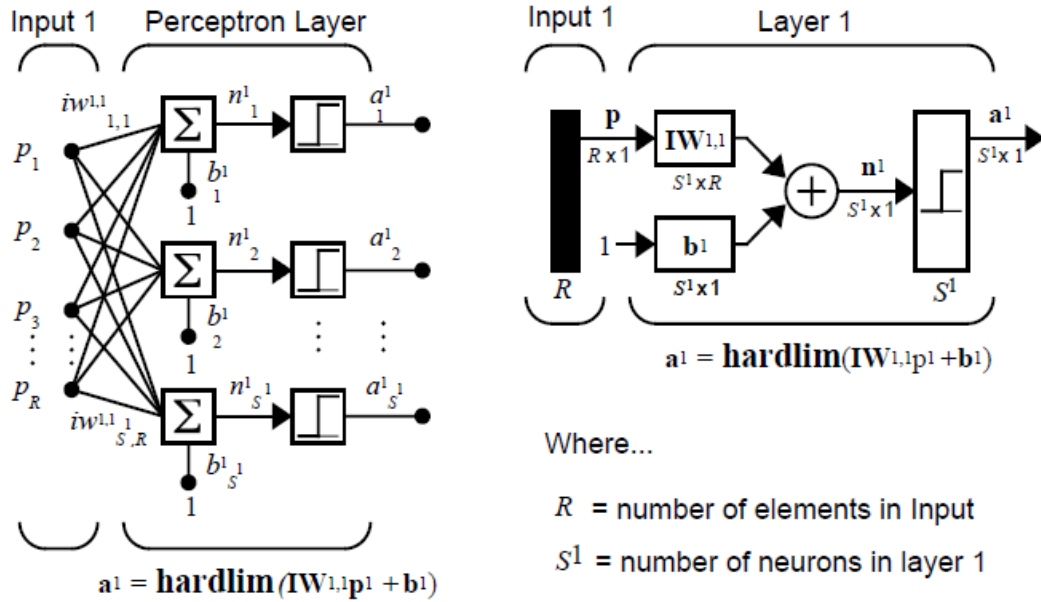
Hard-limit neurons without a bias will always have a classification line going through the origin. *Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin.* The bias allows the decision boundary to be shifted away from the origin as shown in the plot above.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights $w_{i,j}$ as shown below in two forms. As before,

the network indices i and j indicate that $w_{i,j}$ is the strength of the connection from the j th input to the i th neuron.

The perceptron learning rule that we will describe shortly can train only a single layer. Thus, here we will consider only one-layer networks. This restriction places limitations on the computation a perceptron can perform.



Where...

R = number of elements in Input

S^1 = number of neurons in layer 1

Learning Rules

A **learning rule** can be defined as a procedure for modifying the weights and biases of a network. (This procedure may also be referred to as a training algorithm). The learning rule is applied to train the network to perform some tasks. Learning rules fall into two broad categories: supervised learning, and unsupervised learning.

In **supervised learning**, the learning rule is provided with a set of examples (the *training set*) of proper network behavior.

$$\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_Q, t_Q\}$$

where p_Q is an input to the network, and t_Q is the corresponding correct (*target*) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network to move the network outputs closer to the targets. The perceptron discussed in this experiment is trained with supervised learning.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs: $p_1, t_1, p_2, t_2, \dots, p_Q, t_Q$.

where $\mathbf{p}$ is an input to the network and $\mathbf{t}$ is the corresponding correct (target) output. The objective in this type of perceptron learning rule is to reduce the error $\mathbf{e}$, which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response $\mathbf{a}$, and the target vector $\mathbf{t}$. The *perceptron learning rule* (**learnp**) calculates desired changes to the perceptron's weights and biases given an input vector ($\mathbf{p}$), and the associated error ($\mathbf{e}$). The target vector ($\mathbf{t}$) must contain values of either 0 or 1, as Perceptrons (with hardlim transfer function) can only output such values.

The perceptron learning rule can be written in terms of the error $\mathbf{e} = \mathbf{t} - \mathbf{a}$, and the change to be made to the weight vector $\Delta \mathbf{w}$:

$$\Delta \mathbf{w} = (\mathbf{t} - \mathbf{a})\mathbf{p}^T = \mathbf{e}\mathbf{p}^T$$

We can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta \mathbf{b} = (\mathbf{t} - \mathbf{a})(1) = \mathbf{e}$$

The Perceptron Learning Rule can be summarized as follows:

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \mathbf{e}\mathbf{p}^T \quad \text{and} \quad \mathbf{b}_{new} = \mathbf{b}_{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t} - \mathbf{a}$,

After computing the output and the error we can use the function **learnp** to start learning. **learnp** used several arguments and returns the change in weights, what we need to know right now is the following:

$$d\mathbf{w} = \text{learnp}(\mathbf{w}, \mathbf{p}, [], [], [], [], \mathbf{e}, [], [], [], [], [])$$

where $d\mathbf{w}$ is the change of weights.

The process of finding new weights (and biases) can be repeated until there are no errors. Note that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are "linearly separable."

Training (train)

If **sim** and **learnp** are used repeatedly to present inputs to a perceptron, and *to change the perceptron weights and biases according to the error*, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traverse through all of the training input and target vectors is called a *pass*.

The function **train** carries out such a loop of calculation. In each pass the function **train** proceeds through the specified sequence of inputs, calculating the output, error and network adjustment for each input vector in the sequence as the inputs are presented. Note that **train** does not guarantee that the resulting network does its job. The new values of $\mathbf{W}$ and $\mathbf{b}$ must be checked by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully, it can be trained further by calling **train** with the new weights and biases

for more training passes, or the problem can be analyzed to see if it is a suitable problem for the perceptron.

The **train** function trains a network **net** according to **net.trainFcn** and **net.trainParam**. Note that the default training function for networks created with **newp** is **train**. This training function applies the perceptron learning rule in its pure form.

Procedure:

In this procedure we try to learn how to create a perceptron, showing how the simulation works, and illustrate the use of learning and training functions.

1. A perceptron can be created with the function *newp*.

net=newp(PR, S)

where input arguments:

PR is an R-by-2 matrix of minimum and maximum values for R input elements.

S is the number of neurons.

Commonly the **hardlim** function is used in Perceptrons, so it is the default.

- 1.1. Try using the code below to create a perceptron network with a single one-element input vector and one neuron. The range for the single element of the single input vector is [0 2].

net=newp([0 2], S);

- 1.2. Execute the following code to see what network has been created:

which yields:

inputweights =

delays: 0

initFcn: 'initzero'

learn: 1

learnFcn: 'learnp'

learnParam: []

size: [1 1]

userdata: [1x1 struct]

weightFcn: 'dotprod'

where the sub object properties are:

net.inputWeights{i,j}.delays : this property defines a tapped delay line between the *j*th input and its weight to the *i*th layer.

net.inputWeights{i,j}.initFcn: this property defines the function used to initialize the weight matrix. This function can be set to the name of any weight initialization function.

net.inputWeights{i,j}.learn: this property defines whether the weight matrix to the *i*th layer from the *j*th input is to be altered during training and adaption. It can be set

to 0 or 1. It enables or disables the weights learning during calls to either adapt or train.

net.inputWeights{i,j}.learnFcn: this property defines the function used to update the weight matrix. It can be set to the name of any weight learning function.

net.inputWeights{i,j}.learnParam: this property defines the learning parameters and values for the current learning function. of the *i*th layer's weight coming from the *j*th input. The fields of this property depend on the current learning function.

net.inputWeights{i,j}.size(read-only): this property defines the dimensions of the *i*th layer's weight matrix from the *j*th network input. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix. The first element is equal to the size of the *i*th layer. The second element is equal to the product of the length of the weights delay vectors with the size of the *j*th input.

net.inputWeights{i,j}.userdata: this property provides a place for users to add custom information

net.inputWeights{i,j}.weightFcn: this property defines the function used to apply. It can be set to the name of any weight function. The weight function is used when sim is called to simulate the network.

Note that the default learning function is **learnp**. The net input to the **hardlim** transfer function is **dotprod**, which generates the product of the input vector and weight matrix and adds the bias to compute the net input. Also note that the default initialization function, **initzero**, is used to set the initial values of the weights to zero.

1.3. Execute the following code for the biases of the network

biases = net.biases{1};

which gives,

```
biases =  
initFcn: 'initzero'  
learn: 1  
learnFcn: 'learnp'  
learnParam: []  
size: 1  
userdata: [1x1 struct]
```

We can see that the default initialization for the bias is also 0.

2. Simulation (sim): To show how sim works examine the following simple problem:

Suppose we take a perceptron with a single two-element input vector, like that discussed in the *decision boundary* figure.

2.1. Define the network using the following code :

net = newp([-2 2;-2 +2],1);

As noted above, this gives us zero weights and biases.

- 2.2. Now set the two weights and the one bias to **-1, 1** and **1** as they were in the decision boundary figure using the following two lines of code.

```
net.IW{1,1} = [-1 1];  
net.b{1} = [1];
```

- 2.3. To make sure that these parameters were set correctly, check them with

```
net.IW{1,1}  
net.b{1}
```

- 2.4. Now check if the network responds to two signals, one on each side of the Perceptron boundary using the **sim** function with **p1 = [1;1]**, and **p2 = [1;-1]**.

- 2.5. Now present the two inputs in a sequence (**p3 = {[1;1] [1;-1]}**) and check the outputs (are they in sequence also or not?).

3. Initialization (**init**): we can use the function **init** to reset the network weights and biases to their original values.

- 3.1. For instance, start with the following definition of the network

```
net = newp([-2 2;-2 +2],1);
```

Now check its weights and bias with

```
wts = net.IW{1,1}  
bias = net.b{1}
```

- 3.2. Set the weights to the values 3 and 4 and the bias to the value 5 and recheck the weights and bias using the appropriate code to verify that the change has been made.

- 3.3. Use **init** to reset the weights and bias to their original values and check the results.

- 3.4. We can change the way that a perceptron is initialized with **init**. For instance, redefine the network input weights and bias **initFcn** as **rands**, and then apply **init** as shown below.

```
net.newp([-2 2;-2 2],1);  
net.inputweights{1,1}.initFcn = 'rands';  
net.biases{1}.initFcn = 'rands';  
net = init(net);  
wts = net.IW{1,1}  
bias = net.b{1}
```

4. Perceptron learning rule (**learnp**): Now try to start learning with a single neuron, having an input vector with just two elements. Set the bias **b** to 0 and the weights **w** to 1 and -0.8. The input target pair is given by **p=[1;2]**, **t = [1]**. Find the change in weights and bias using the function **learnp**.

```
dw=learnp(w,P,[],[],[],[],e,[],[],[],[])  
w=w+dw
```

5. To illustrate the training procedure, consider a one neuron perceptron with a single vector input having two elements. Suppose that we have the following single vector input,

$$\{p_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0\} \{p_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1\} \{p_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0\} \{p_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1\}$$

The initial weights $w = [0 \ 0]$ and initial bias $b = 0$. try to solve the problem, using the following code:

- 5.1. Set epochs to 1, so that train will go through the input vectors just one time.

```
net = newp([-2 2;-2 +2],1);  
net = init(net);  
wts = net.IW{1,1}  
bias = net.b{1}  
net.trainParam.epochs = 1;  
net.layers{1}.transferFcn='hardlims';
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]  
t = [0 1 0 1]
```

Now train the network with

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

The new weights and bias are $w = -3 \ -1 \ b = 0$

Finally simulate the trained network for each of the inputs.

```
a = sim(net,p)  
a = 0 0 1 1
```

The outputs do not yet equal the targets, so we need to train the network for more than one pass.

- 5.2. Set epochs to 4. This run gives the following results.

TRAINC, Performance goal met.

The final weights and bias are

$w = -2 \ -3 \ b = 1$

The simulated output and errors for the various inputs are

```
a = 0      1.00      0      1.00  
error = [a(1)-t(1) a(2)-t(2) a(3)-t(3) a(4)-t(4)]  
error = 0 0 0 0
```

The network was trained by the time the inputs were presented on the third epoch. Thus, we have checked that the training procedure was successful. The network converged and produced the correct target outputs for the four input vectors.

6. In this step, we will use code program to execute Perceptron model without using MATLAB built-in function, the results should be identical.

```
clc; clear all; close all;
x=[2 1 -2 -1; 2 -2 2 1]; t=[0 1 0 1];
b=0; w=[0;0]; alpha=1;
for it =1 :100
    for i=1:length(x)
        net=x(:,i)*w+b;
        net(i)=hardlim(net);
        ynet(i)=net(i);
        error=t(i)-ynet(i);
        if error~=0
            b=b+alpha*t(i);
            w=w+alpha*t(i)*x(:,i);
        else
            b=b;
            w=w;
        end
    end
end

acc(it,:)=t-ynet;
if sum(acc(it,:))==0
    break;
end
end
m=-(w(1)/w(2));
c=-b/w(2)

display('final weight');w
display('final bias');b
plot(x(1,1),x(2,1),'or'); hold on
plot(x(1,2),x(2,2),'or'); hold on
plot(x(1,3),x(2,3),'or'); hold on
plot(x(1,4),x(2,4),'or'); hold on
axis([-3 3 -3 3])
x1=linspace(-3,3,100);
x2=m*x1-c;
plot(x2,x1,'k')
```

Results:

2.3 The values of the weights and bias

net.IW{1,1} net.b{1}

ans = -1 1 ans = 1

2.4 The code and results:

p1 = [1;1];

a1 = sim(net,p1)

result:

a1 = 1

and

p2 = [1;-1]

a2 = sim(net,p2)

result:

a2 = 0

The perceptron classified the two inputs correctly, (i.e., one on each side of the perceptron boundary

2.5 The code and results:

p3 = {[1;1] [1;-1]};

a3 = sim(net,p3)

result:

a3 = [1] [0]

Which means that the outputs are in sequence

3.1 The weights and bias:

wts = 0 0 bias = 0

3.2 The code and results:

net.IW{1,1} = [3,4];

net.b{1} = 5;

results:

wts = 3 4 bias = 5

3.3 The code and results:

net = init(net);

results:

wts = 0 0 bias = 0

3.4 We can see that the weights and bias have been given random numbers such as:

wts = 0.2309 0.5839 biases = -0.1106

4. The code and results of perceptron learning rule learnp:

net = newp([-2 2;-2 +2],1);

net.b{1} = [0];

w = [1 -0.8];

```
net.IW{1,1} = w;  
p = [1; 2];  
t = [1];  
a = sim(net,p)  
e = t-a  
dw = learnnp(w,p,[],[],[],[],e,[],[],[],[],[])  
w = w + dw  
results:
```

a = 0 e = 1 dw = 1 2 w = 2.0000 1.2000

Experiment No. (5) Design of Adaptive Linear Filter Using ADALINE

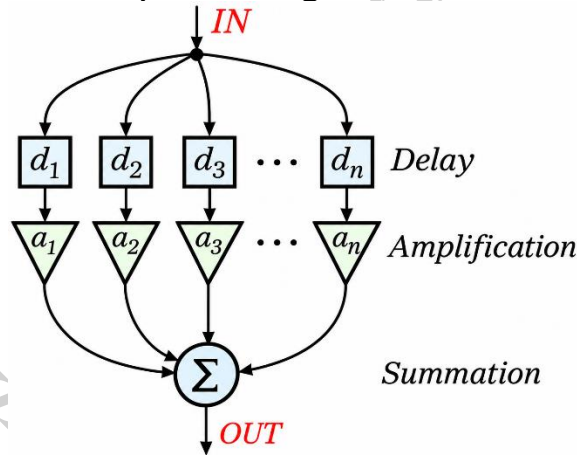
Object:

The object of this experiment is to design adaptive filter by using ADALINE neural network and show how to make adaptive training.

Theory:

An **adaptive filter** is a filter that *self-adjusts* its transfer function according to an optimizing algorithm. Because of the complexity of the optimizing algorithms, most adaptive filters are digital filters that perform digital signal processing and adapt their performance based on the input signal. By way of contrast, a non-adaptive filter has static filter coefficients.

A **digital filter** is a system that performs mathematical operations on a sampled, discrete-time signal to reduce or enhance certain aspects of that signal. The figure below shows a general finite impulse response (FIR) filter with n stages, each with an independent delay, d_i , and amplification gain, a_i .



A diagram of a general FIR filter

The output of a discrete-time FIR filters can be defined in terms of its inputs by the following equation:

$$y[n] = b_0x[n] + b_1x[n - 1] + \dots + b_Nx[n - N]$$

where $x[n]$ is the input signal, $y[n]$ is the output signal, b_i are the **filter coefficients**, also known as **tap weights**, and N is the filter order (an N th-order filter has $(N + 1)$ terms). These are commonly referred to as **taps** (the number of inputs). This equation can also be expressed as a convolution of the coefficient sequence b_i with the input signal:

$$y[n] = \sum_{i=0}^N b_i x[n - i]$$

That is, the filter output is a weighted sum of the current and a finite number of previous values of the input.

Digital filters are commonplace and an essential element of everyday electronics such as radios, cell phones, and stereo receivers.

Adaptive Linear Neuron networks

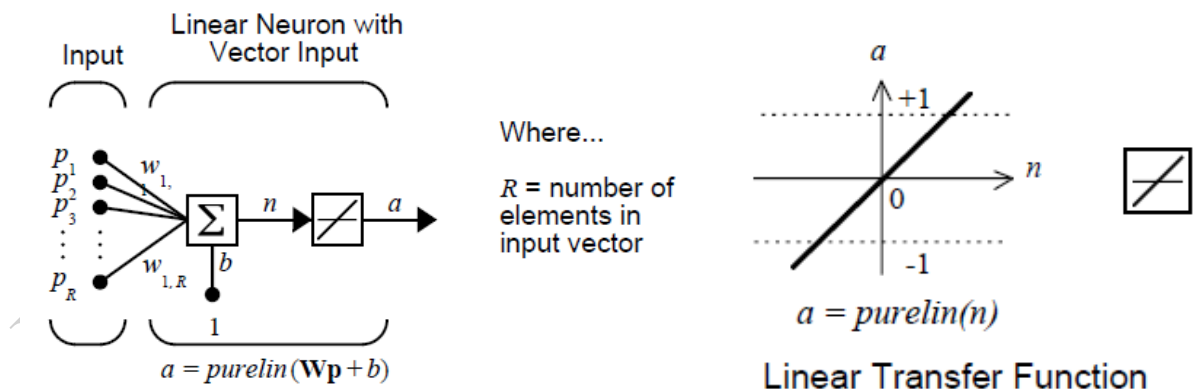
The ADALINE networks discussed in this experiment are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their output to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can only solve linearly separable problems. However, here we will make use of the LMS (**Least Mean Squares**) learning rule, which is *much more powerful than the perceptron learning rule*. The **LMS or Widrow-Hoff** learning rule minimizes the mean square error and, thus, moves the decision boundaries as far as it can from the training patterns.

In this experiment, we design an adaptive linear system that responds to changes in its environment as it is operating. **Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors.** Networks of this sort are often used in error cancellation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements.

Linear Neuron Model

A linear neuron with R inputs (simple ADALINE network) is shown below:



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a *linear transfer function*, which we name **purelin**. The linear transfer function calculates the neuron's output by simply returning the value passed to it.

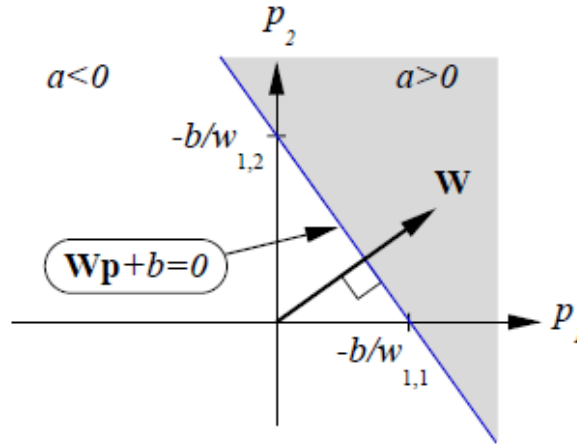
This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

$$a = \text{purelin}(n) = \text{pureline}(Wp + b) = Wp + b$$

Consider a single ADALINE with two inputs. The weight matrix W in this case has only one row. The network output is:

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $Wp + b = 0$ specifies such a decision boundary as shown below:



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories.

Mean Square Error

The least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{p_1, t_1\}, \{p_1, t_1\}, \dots, \{p_Q, t_Q\}$$

Here p_Q is an input to the network, and t_Q is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. We want to minimize the average sum of these errors.

$$mse = \frac{1}{Q} \sum_{k=1}^Q e(k)^2 = \frac{1}{Q} \sum_{k=1}^Q (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error. Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance

index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

Adaptive networks will use the LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

$$\mathbf{w}(k+1) = \mathbf{w}(k) + 2\alpha e(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha e(k)$$

Fortunately, we have a toolbox function *learnwh* that does all of the calculation for us. It calculates the change in weights as:

$$\Delta \mathbf{w} = \alpha \mathbf{e} \mathbf{p}^T$$

and the bias change as:

$$\Delta \mathbf{b} = \alpha \mathbf{e}$$

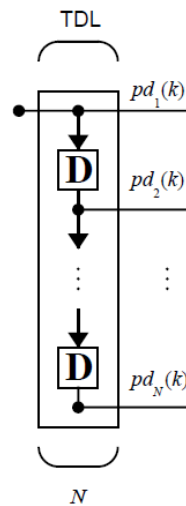
Constant 2, shown in above equations, has been absorbed into the learning rate value.

Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. Nevertheless, the ADALINE has been and is today one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

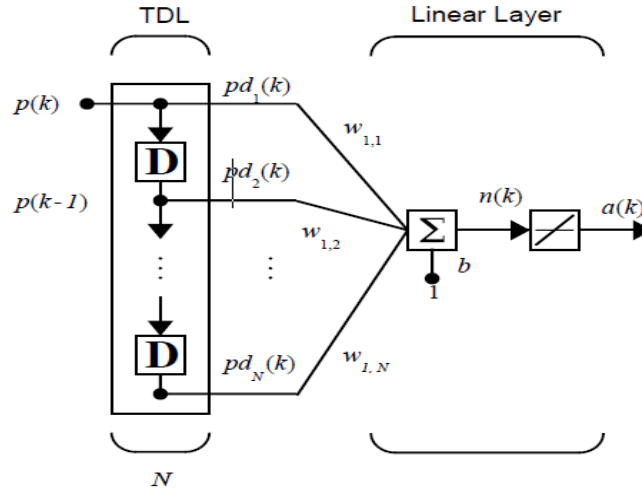
Tapped Delay Line

We need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown below. There the input signal enters from the left and passes through $N-1$ delays. The output of the **tapped delay line** (TDL) is an N -dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Adaptive Filter

We can combine a tapped delay line with an ADALINE network to create the adaptive filter shown below:



The output of the filter is given by,

$$a = \text{purelin}(Wp + b) = \sum_{i=1}^R w_{1,i} a(k - i + 1) + b$$

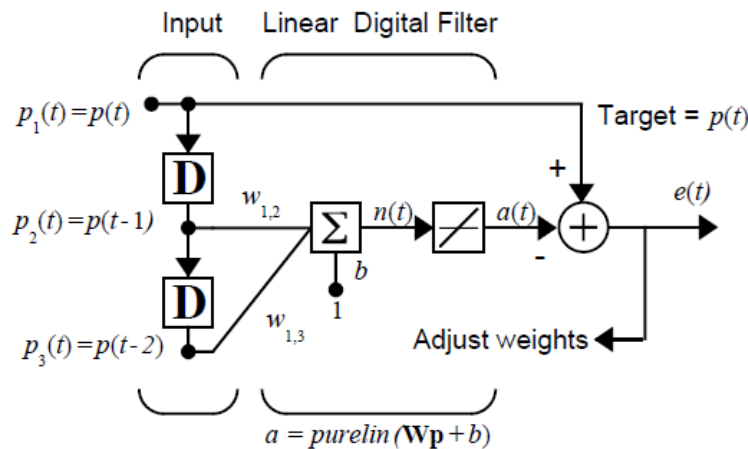
The network shown above is referred to in the digital signal processing field as a **finite impulse response (FIR) filter**.

Procedure

In this experiment we are going to generate and simulate an adaptive network which represents an adaptive filter example.

This experiment introduces the function **adapt**, which changes the weights and biases of a network incrementally during training. Let us take a look at the code that we use to generate and simulate such an adaptive network.

For the example shown in the figure:



Predictive Filter: $a(t)$ is approximation to $p(t)$

1. Define the input and output data, use the following commands:

The input variables are composed of two time series vectors, as follows:

```
close all, clear, clc, format compact
dt = 0.01;      % time step [seconds]
t1 = 0 : dt : 3; % first time vector [seconds]
t2 = 3+dt : dt : 6; % second time vector [seconds]
t = [t1 t2];    % complete time vector [seconds]
```

Time series (y), output vector, is composed of two different segments with different amplitude and frequency.

```
Y = [sin(4.1*pi*t1) .8*sin(8.3*pi*t2)];
```

2. Plot the time series signal y along with time t, as follows:

```
plot(t,y,'.-')
xlabel('Time [sec]');
ylabel('Target Signal');
grid on
ylim([-1.2 1.2])
p = get(gcf,'Position');
set(gcf,'Position',[p(1:3) p(4)/2])
```

3. Prepare data for neural network

For sequential vectors, the order in which the vectors appear is important. Therefore, convert the concurrent vector to sequential one, as follows:

```
Y = con2seq(y);
```

4. Define ADALINE neural network

```
inputDelays = 1:5; % delayed inputs to be used
learning_rate = 0.2; % learning rate
```

To define ADALINE network

```
net = linearlayer(inputDelays,learning_rate);
```

To see ADALINE network, use the following commands:

```
view(net);
```

5. ADALINE neural network is adaptively trained to predict the next step of the time series.

```
[net,P,E] = adapt(net,Y,Y);
```

To display trained ADALINE network

```
view(net)
```

6. To see the final network parameters, weights (iw), and biases (b), after adaptation:

```
iw = net.IW{1}
```

```
b = net.b{1}
```

7. Return the vector to concurrent one for visualization, as follows:

```
p = seq2con(P); p = p{1};
```

```
e = seq2con(E); e = e{1};
```

8. Plot the predicted output and the error, as follows:

```
figure
```

```
% first graph
```

```
subplot(211)
```

```
plot(t,y, t,p,'r--');
```

```
legend('Original','Prediction')
```

```
grid on
```

```
xlabel('Time [sec]');
```

```
ylabel('Target Signal');
```

```
ylim([-1.2 1.2])
```

```
% second graph
```

```
subplot(212)
```

```
plot(t,e);
```

```
grid on
```

```
legend('Prediction error')
```

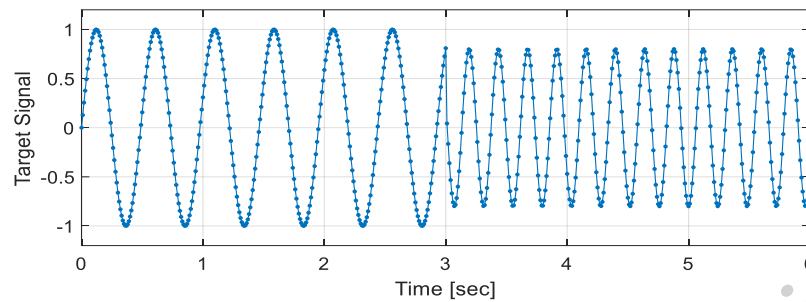
```
xlabel('Time [sec]');
```

```
ylabel('Error');
```

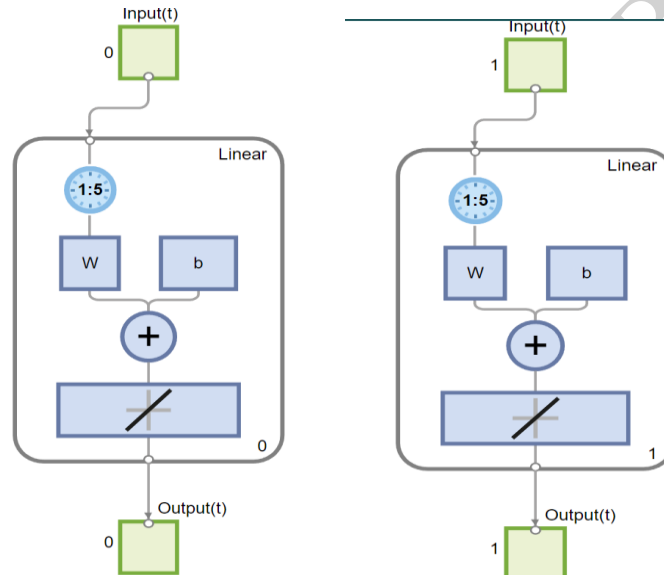
```
ylim([-1.2 1.2])
```

Results:

- Step 2, the output is



- Step 4 and 5, to see the ADALINE network before and after training:

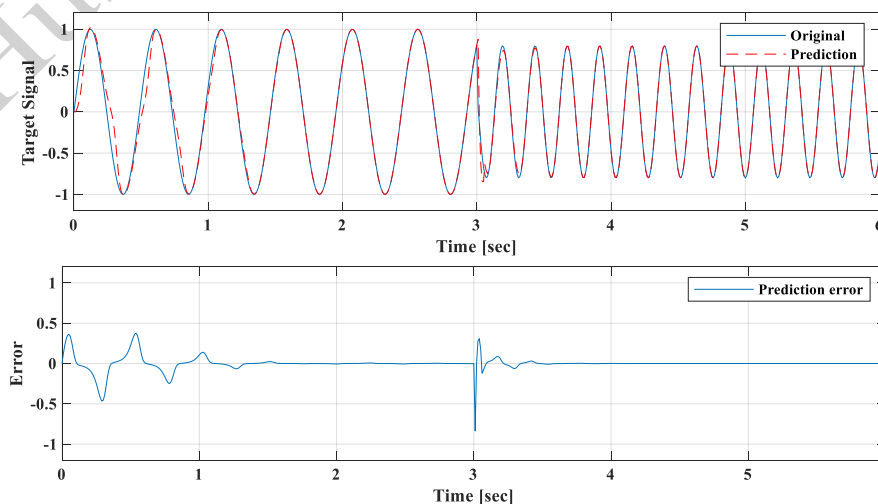


- Step 6, to see the weights and bias values:

$w = 0.7179 \quad 0.4229 \quad 0.1552 \quad -0.1203 \quad -0.4159$

$b = -1.2520e-08$

- Step 6, plot the predicted output and the error



Experiment No. (6) Backpropagation Neural Networks

Object:

The primary objective of this experiment is to show how to use the backpropagation training functions in training multilayer feedforward neural networks that can solve specific problems.

Theory:

Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as it is defined. Networks with biases, a sigmoid layer, and a linear output layer can approximate any function with a finite number of discontinuities.

Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. **The term *backpropagation* refers to the way the gradient is computed for nonlinear multilayer networks.** There are several variations on the basic algorithm that are based on other standard optimization techniques, such as **conjugate gradient and Newton methods.**

There are generally four steps in the backpropagation training process:

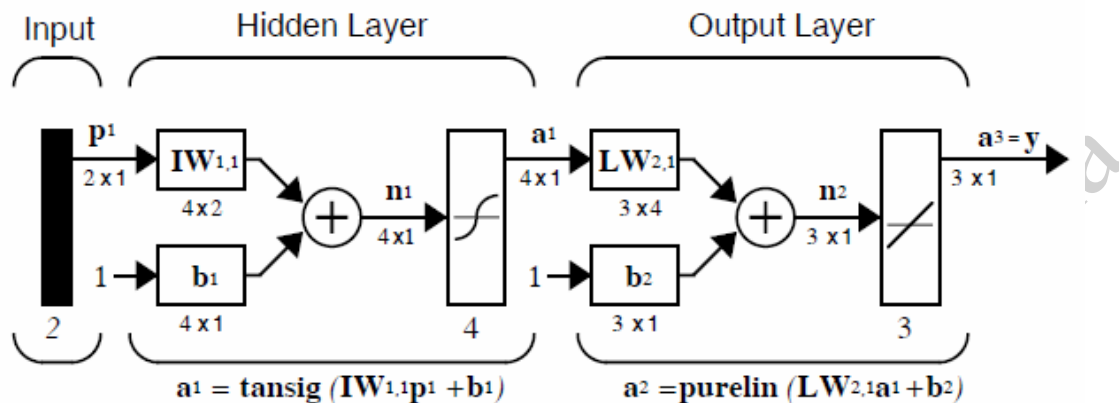
1. Assemble the training data
2. Create the network object
3. Train the network
4. Simulate the network response to new inputs

Architecture

The multilayer feedforward network represents the most used network with the backpropagation algorithm. The routines in the Neural Network Toolbox can be used to train more general networks.

Feedforward networks often have one or more hidden layers of ***sigmoid neurons*** followed by an output layer of ***linear neurons***. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range -1 to $+1$. On the other hand, if it is wanted to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as $\log\text{sig}$).

The network shown in the figure below can be used as a general function approximator. It can approximate any function with a finite number of discontinuities, arbitrarily well, given sufficient neurons in the hidden layer.



Multilayer networks often use the log-sigmoid transfer function **logsig**. The function logsig generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity. Alternatively, multilayer networks may use the tan-sigmoid transfer function **tansig**.

Occasionally, the linear transfer function **purelin** is used in backpropagation networks. If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

In backpropagation it is important to be able to calculate the derivatives of any transfer functions used. Each of the transfer functions above, tansig, logsig, and purelin, have a corresponding derivative function: *dtansig*, *dlogsig*, and *dpurelin*. To get the name of a transfer function's associated derivative function, call the transfer function with the string 'deriv'.

tansig('deriv')

ans = dtansig

The three transfer functions described here are the most used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired.

Creating a Network (newff)

The first step in training a feedforward network is to create the network object. The function **newff** creates a feedforward network. It requires four inputs and returns the network object:

Net = newff(PR, [S1 S2, ...,SNI], {TF1 TF2...TFNI}, BTF) takes,

- The first input **PR** is an R by 2 matrix of minimum and maximum values for each of the R elements of the input vector.

- The second input **Si** is an array containing the sizes of each layer (S1, S2, ..., SN1), for N1 layers.
- The third input **TFi** is a cell array containing the names of the transfer functions to be used in each layer.
- The last input **BTF** contains the name of the training function to be used. and returns an N layer feed-forward backpropagation network. This command creates the network object and also initializes the weights and biases of the network.

Training

Once the network weights and biases have been initialized, the network is ready for training. The training process requires a set of examples of proper network behavior - network inputs *p* and target outputs *t*. During training the weights and biases of the network are iteratively adjusted to minimize the network performance function **net.performFcn**. The default performance function for feedforward networks is mean square error **mse** (the average squared error between the network outputs (*a*) and the target outputs *t*).

Backpropagation Algorithm

There are many variations of the backpropagation algorithm. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly (*the negative of the gradient*).

One iteration of this algorithm can be written,

$$\mathbf{X}_{k+1} = \mathbf{X}_k - \alpha_k \mathbf{g}_k$$

where $\mathbf{X}_k$ is a vector of current weights and biases, $\mathbf{g}_k$ is the current gradient, and α_k is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: **incremental mode and batch mode**. In the **incremental mode**, the **gradient** is computed and the weights are updated after each input is applied to the network. In the **batch mode** all of the inputs are applied to the network before the weights are updated.

Batch Training (train)

In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in weights and biases.

Batch Gradient Descent (traingd): The batch steepest descent training function is **traingd**. The weights and biases are updated in the direction of the negative gradient

of the performance function. If you want to train a network using batch steepest descent, you should set the network **trainFcn** to **traingd** and then call the function train. There is only one training function associated with a given network.

There are seven training parameters associated with traingd:

epochs, show, goal, time, min_grad, max_fail, and lr.

- The learning rate **lr** is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge.
- The training status is displayed for every show iteration of the algorithm. (If show is set to NaN, then the training status never displays.)
- The other parameters determine when the training stops. The training stops if the ***number of iterations exceeds epochs***, if the ***performance function drops below goal***, if the ***magnitude of the gradient is less than mingrad***, or if the ***training time is longer than time seconds***. The ***max_fail*** is normally used to stop training early if the ***network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row***.

For batch training, all of the input vectors are placed in one matrix.

Batch Gradient Descent with Momentum (traingdm): In addition to traingd, there is another batch algorithm for feedforward networks that often provides faster convergence ***traingdm, steepest descent with momentum***. ***Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface.*** Acting like a low-pass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum.

Momentum can be added to backpropagation learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the backpropagation rule. The magnitude of the effect that the last weight change is allowed to have is mediated by a ***momentum constant, mc, which can be any number between 0 and 1***. When the momentum constant is 0, a weight change is based solely on the gradient. When the momentum constant is 1, the new weight change is set to equal the last weight change, and the gradient is simply ignored. The gradient is computed by summing the gradients calculated at each training example, and the weights and biases are only updated after all training examples have been presented.

If the new performance function on a given iteration exceeds the performance function on a previous iteration by more than a predefined ratio ***max_perf_inc*** (***typically 1.04***), the new weights and biases are discarded, and the momentum coefficient *mc* is set to zero.

The batch form of gradient descent with momentum is invoked using the training function ***traingdm***. The ***traingdm*** function is invoked using the same steps shown above for the ***traingd*** function, except that the ***mc***, and ***max_perf_inc*** learning parameters can all be set.

Procedure

1. Batch Gradient Descent

- 1.1. Use the following vectors as a training set of input ***p*** and target ***t***
***p* = [-1 -1 2 2;0 5 0 5];**
***t* = [-1 -1 1 1];**
- 1.2. Next, create the feedforward network having, three neurons in the hidden layer and single neuron in the output layer. Use the function ***minmax*** to determine the range of the inputs to be used in creating the network.
- 1.3. Modify the following training parameters.
***net.trainParam.show* = 50;**
***net.trainParam.lr* = 0.05;**
***net.trainParam.epochs* = 300;**
***net.trainParam.goal* = 1e-5;**

Note: If we want to use the default training parameters, the above commands are not necessary.

- 1.4. Train the network using the following code, and check for the performance goal:
[*net*,*tr*]=train(*net*,*p*,*t*);

The training record ***tr*** contains information about the progress of training, such as, ***tr.epochs*** and ***tr.perf***.

- 1.5. Simulate the network to obtain its response to the inputs in the training set.
Record the outputs.
***a* = sim(*net*,*p*)**

2. Batch Gradient Descent with Momentum:

Recreate the previous network and retrain it using gradient descent with momentum, then simulate it to the inputs in the training set. The training parameters for ***traingdm*** are the same as those for ***traingd***, with the addition of the momentum factor ***mc*** and the maximum performance increase ***max_perf_inc***.
***net.trainParam.mc* = 0.9;**

- Note that since we reinitialized the weights and biases before training (by calling `newff` again), we obtain a different mean square error than we did using `traingd`. If we were to reinitialize and train again using `traingdm` and **Levenberg-Marquardt backpropagation (`trainlm`)**, we would get yet a different mean square error. The random choice of initial weights and biases will affect the performance of the algorithm.
- 3. Compare the performance (**Required number of epochs, MSE, Gradient**, the simulation output) for the above two algorithms of batch training. Then plot `tr.epoch` vs. `tr.perf`. label the x-axis '**Epoch**', and the y-axis '**Performance**'.

Note: In order to compare the performance of different algorithms, you should test each using several different sets of initial weights and biases, (consider three sets).

4. Code a computer program to implement a backpropagation neural network with one hidden layer. Use a bias on each hidden unit and each output unit. Use **10 input** units, **4 hidden** units, and **2 output** units. Use the **hyperbolic tangent sigmoid transfer function**. The **momentum coefficient** is **0.9**. For each test case, print the initial weights, final weights, learning rate, number of training epochs, and network response to each input pattern at the end of the training. The input patterns are:

$s(1) = 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9$	$s(5) = 3 \ 8 \ 2 \ 7 \ 1 \ 6 \ 0 \ 5 \ 9 \ 4$
$s(2) = 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \ 0$	$s(6) = 1 \ 6 \ 0 \ 7 \ 4 \ 8 \ 3 \ 9 \ 2 \ 5$
$s(3) = 0 \ 9 \ 1 \ 8 \ 2 \ 7 \ 3 \ 6 \ 4 \ 5$	$s(7) = 2 \ 1 \ 3 \ 0 \ 4 \ 9 \ 5 \ 8 \ 6 \ 7$
$s(4) = 4 \ 5 \ 6 \ 3 \ 2 \ 7 \ 1 \ 8 \ 0 \ 9$	$s(8) = 9 \ 4 \ 0 \ 5 \ 1 \ 6 \ 2 \ 7 \ 3 \ 8$

The corresponding output (target) patterns are:

$t(1) = -1 \ -1$	$t(5) = 1 \ 1$
$t(2) = 1 \ 1$	$t(6) = 1 \ -1$
$t(3) = -1 \ 1$	$t(7) = -1 \ 1$
$t(4) = 1 \ -1$	$t(8) = -1 \ -1$

Use a **learning rate** of (i) **0.05** and (ii) **0.5**. For each learning rate, perform **5000** and **10000** epochs of training.

5. Execute the Backpropagation method to represent the XOR problem using MATLAB code program using step descent gradient (SGD) method, then validate the using Backpropagation built-in function.

```
clc; clear all;
X = [0 0; 0 1; 1 0; 1 1];
D = [0; 1; 1; 0];
W1 = 2*rand(4, 2) - 1; % Matrix of weights for inputs
W2 = 2*rand(1, 4) - 1; % Matrix of weights for output
for epoch = 1:10000 % train
[W1 W2] = BackpropXOR(W1, W2, X, D); % Function of Backpropagation for XOR
end
```

```

N = 4; % inference
for k = 1:N
    x = X(k, :);
    v1 = W1*x;           % W1*x
    y1 = logsig(v1);
    v = W2*y1;           % W2*y
    y(k) = logsig(v)
end
bar(1:N,D)
hold on
bar(1:N,y,0.4,'k')

```

% Function of Backpropagation for XOR in different M.file

```

function [W1, W2] = BackpropXOR(W1, W2, X, D)
alpha = 0.9;
N = 4;
for k = 1:N
    x = X(k, :);
    d = D(k);
    v1 = W1*x;           % W1*x
    y1 = logsig(v1);
    v = W2*y1;           % W2*y
    y = logsig(v);
    e = d - y;           % e = d-y % error at output layer
    delta = y.*(1-y).*e; %  $\delta = \varphi'(v)e$ , where  $\varphi'$  is derivative of logsig function
    e1 = W2'*delta;
    delta1 = y1.*(1-y1).*e1;
    dW1 = alpha*delta1*x'; %  $\Delta w = \alpha \delta x$ 
    W1 = W1 + dW1;         %  $w_{new} = w_{old} + \Delta w$ 
    dW2 = alpha*delta*y1';
    W2 = W2 + dW2;
end
end

```

Results:

1. *Batch Gradient Descent*

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingd');
% net=newff(minmax(p),[3,1],{'tansig','purelin'}, trainlm);
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

Now we are ready to train the network.

```
[net,tr]=train(net,p,t);
TRAINGD, Epoch 0/300, MSE 1.59423/1e-05, Gradient 2.76799/ 1e-10
TRAINGD, Epoch 50/300, MSE 0.00236382/1e-05, Gradient 0.0495292/1e-10
TRAINGD, Epoch 100/300, MSE 0.000435947/1e-05, Gradient 0.0161202/1e-10
TRAINGD, Epoch 150/300, MSE 8.68462e-05/1e-05, Gradient 0.00769588/1e-10
TRAINGD, Epoch 200/300, MSE 1.45042e-05/1e-05, Gradient 0.00325667/1e-10
TRAINGD, Epoch 211/300, MSE 9.64816e-06/1e-05, Gradient 0.00266775/1e-10
TRAINGD, Performance goal met.
```

Now the trained network can be simulated to obtain its response to the inputs in the training set.

```
a = sim(net,p)
a =
-1.0010    -0.9989    1.0018    0.9985
```

2. *Batch Gradient Descent with Momentum:*

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingdm');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
TRAINGDM, Epoch 0/300, MSE 3.6913/1e-05, Gradient 4.54729/ 1e-10
TRAINGDM, Epoch 50/300, MSE 0.00532188/1e-05, Gradient 0.213222/1e-10
TRAINGDM, Epoch 100/300, MSE 6.34868e-05/1e-05, Gradient 0.0409749/1e-10
TRAINGDM, Epoch 114/300, MSE 9.06235e-06/1e-05, Gradient 0.00908756/1e-10
TRAINGDM, Performance goal met.
```

```
a = sim(net,p)
a =
-1.0026    -1.0044    0.9969    0.9992
```

3. *Plotting*

```
plot(tr.epoch,tr.perf)
% legend('Training','Validation','Test',-1);
```

ylabel('Performance'); xlabel('Epoch')

4. Code

```
P=[0 9 0 4 3 1 2 9;1 8 9 5 8 6 1 4;2 7 1 6 2 0 3 0; 3 6 8 3 7 7 0 5;4 5 2 2 7 7 0 5; ...
5 4 7 7 6 8 9 6;6 3 3 1 0 3 5 2;7 2 6 8 5 9 8 7; 8 1 4 0 9 2 6 3;9 0 5 9 4 5 7 8]
t=[ -1 1 -1 1 1 1 -1 -1; -1 1 1 -1 1 -1 1 -1]
net=newff(minmax(p),[4,2],{'tansig','tansig'},'traingdm');
net.trainParam.show = 200;
net.trainParam.lr = 0.05;→ 0.5
net.trainParam.mc = 0.9;
net.trainParam.epochs = 5000; → 10000
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
a = sim(net,p)
```

5. Experimental 12 back propagation

```
P = [0 1 0 1; 0 0 1 1];
T = [0 1 1 0];
net = newff(minmax(P),[10 10 1],{'tansig' 'tansig' 'purelin'},'traingd');
Y = sim(net,P);
net.trainParam.show=50;
net.trainParam.lr=0.05;
net.trainParam.epochs = 10;
net.trainParam.goal=1e-5;
net.trainParam.mc=0.9;
[net,tr] = train(net,P,T);
Y2= sim(net,P)
% hold on
plot(P,T,P,Y,'o')
figure
plot(tr.epoch,tr.perf)
ylabel('performance');xlabel('Epoch')
```

Experiment No. (7) Convolutional Neural Network (CNN)

Object:

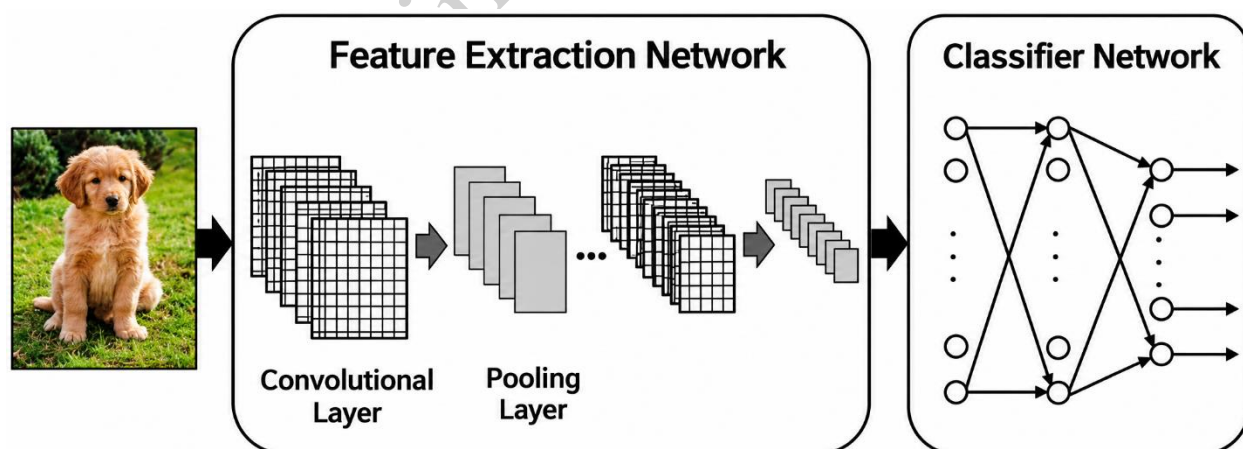
This experiment introduces the convolutional neural network (ConvNet), which is a deep neural network specialized for image recognition. This technique exemplifies how significant the improvement of the deep layers is for information (images) processing.

Theory:

ConvNet is an old technique, which was developed in the 1980s and 1990s. However, it has been forgotten for a while, as it was impractical for real-world applications with complicated images. Since 2012 when it was dramatically revived, ConvNet conquered most computer vision fields and is growing at a rapid pace. A *convolutional neural network (CNN)* is a deep neural network designed to learn spatial features from structured arrays such as images. Instead of manually defining image descriptors, the network learns filters directly from data. Early layers usually respond to local patterns, while deeper layers combine those responses into task-specific representations.

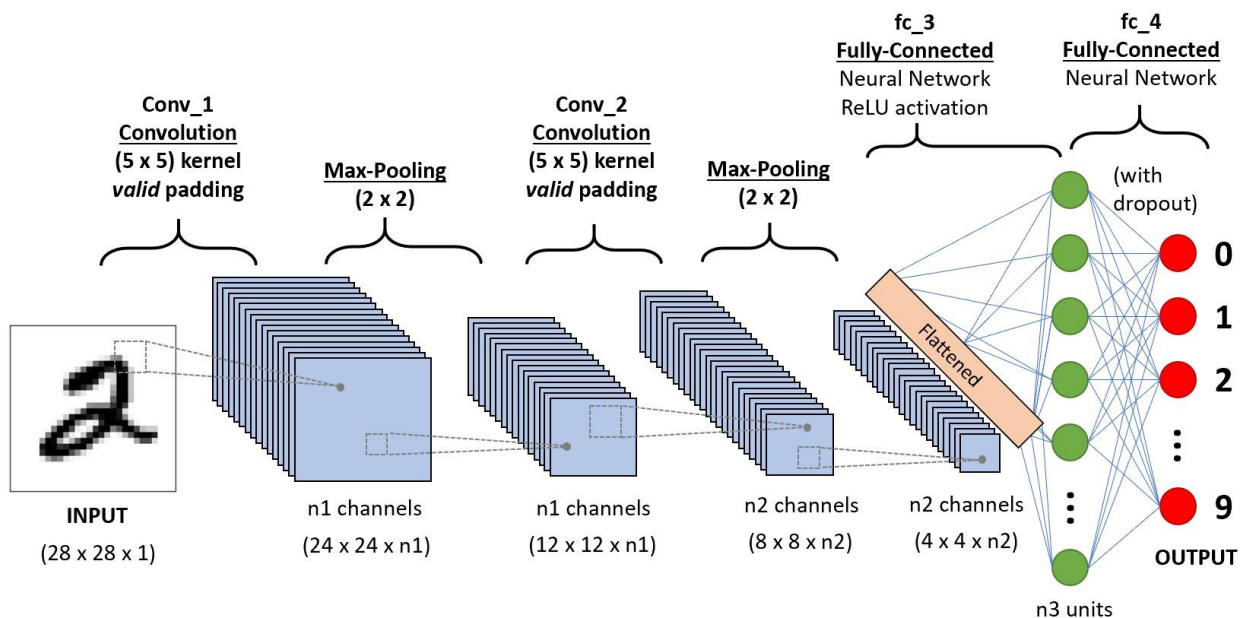
CNN Architecture:

ConvNet consists of a neural network that extracts features of the input image and another neural network that classifies the feature image. Figure below shows the typical architecture of ConvNet.



The input image enters into the feature extraction network. The extracted feature signals enter the classification neural network. The classification neural network then operates based on the features of the image and generates the output.

The **feature extraction** neural network consists of piles of **convolutional layer** and **pooling layer** pairs. The **convolution layer**, as its name implies, converts the image using the convolution operation. It can be thought of as a collection of digital filters. The **pooling layer** combines the neighboring pixels into a single pixel. Therefore, the **pooling layer reduces the dimension of the image**. As the primary concern of ConvNet is the image; the operations of the convolution and pooling layers are conceptually in a two-dimensional plane. This is one of the differences between ConvNet and other neural networks. The **classification network** usually employs the ordinary multiclass classification neural network.



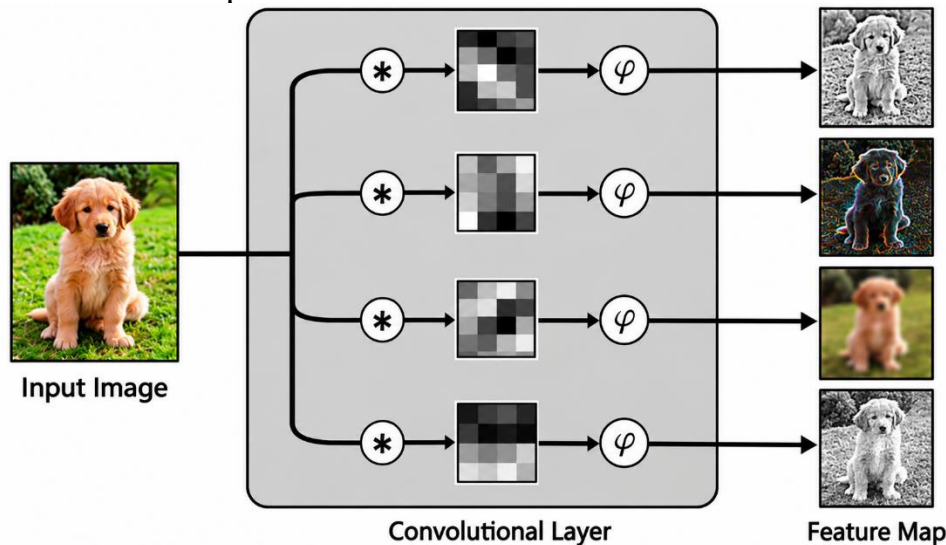
CNN consists of 3 main layers. Those are Convolution layer, Pooling layer, and fully connected layer or dense layer.

Convolution layer

The convolution layer generates new images called *feature maps*. The feature map accentuates the unique features of the original image. The convolution layer operates in a very different way compared to the other neural network layers. This layer does not employ connection weights and a weighted sum. Instead, it contains filters that convert images. We will call these filters **convolution filters**. The process of inputting the image through the convolution filters yields the feature map.

Figure below shows the process of the *convolution layer*, where the circled * mark denotes the **convolution operation**, and the ϕ mark is the **activation function**. The square grayscale icons between these operators indicate the convolution filters. The convolution layer generates the same number of feature maps as the convolution

filters. Therefore, for instance, if the convolution layer contains four filters, it will generate four feature maps.



The filters of the convolution layer are two-dimensional matrices. They usually come in 5×5 or 3×3 matrices, and even 1×1 convolution filter has been used in recent applications. As addressed in the previous section, the values of the filter matrix are determined through the training process. Therefore, these values are continuously trained throughout the training process. This aspect is similar to the updating process of the connection weights of the ordinary neural network.

The convolution operation begins at the upper-left corner of the submatrix that is the same size as the convolution filter.

1	1	1	3
4	6	4	8
30	0	1	5
0	2	2	4

*

1	0
0	1

=

7		

The **convolution operation** is the sum of the products of the elements that are located in the same positions of the two matrices. The result of 7 in Figure is calculated as:

$$(1 \times 1) + (1 \times 0) + (4 \times 0) + (6 \times 1) = 7$$

The element of (3, 1) of the map shows the greatest value. What happened to this cell? This value is the result of the convolution operation shown in Figure below.

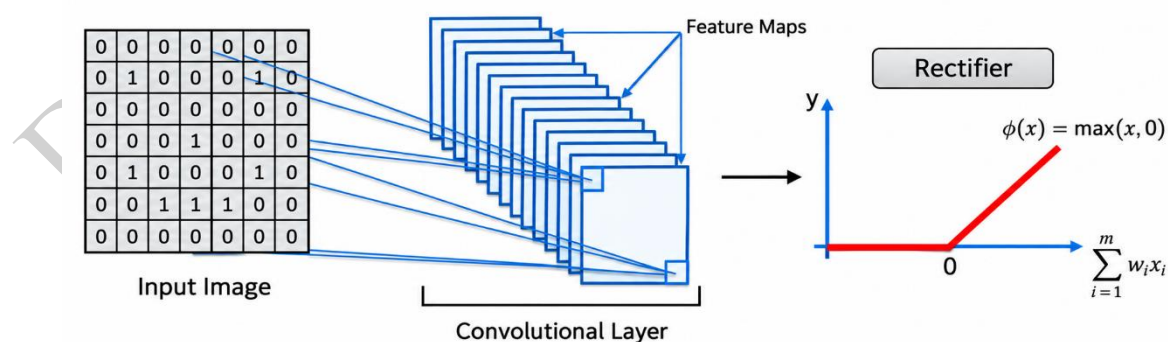
$$\begin{bmatrix} 1 & 1 & 1 & 3 \\ 4 & 6 & 4 & 8 \\ 30 & 0 & 1 & 5 \\ 0 & 2 & 2 & 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 5 & 9 \\ 4 & 7 & 9 \\ 32 & 2 & 5 \end{bmatrix}$$

It is noticeable from the figure that the submatrix of the image matches the convolution filter; *both are diagonal matrices with significant numbers on the same cells*. The convolution operation yields large values when the input matches the filter. first convolution operation, the values in the elements of this feature map depend on whether the image matrix matches the convolution filter or not.

In summary, the convolution layer operates the convolution filters on the Input image and produces the **feature maps**. It uses mathematical operation which takes two inputs such as *image matrix* and a *kernel* or *filter*. The features that are extracted in the convolution layer determined by the trained convolution filters. Therefore, the features that the convolution layer extracts vary depending on which convolution filter is used.

The **feature map** that the convolution filter creates is processed through the **activation function** before the layer yields the output. The activation function of the convolution layer is identical to that of the ordinary neural network. Although the Rectified-Linear Unit (**ReLU**) function is used in most of the recent applications, the sigmoid function and the tanh function are often employed as well

The ReLU layer will determine whether an input node will 'fire' given the input data. The ReLU layer allows the convnet to account for situations in which the relationship between the pixel value inputs and the convnet output is not linear.

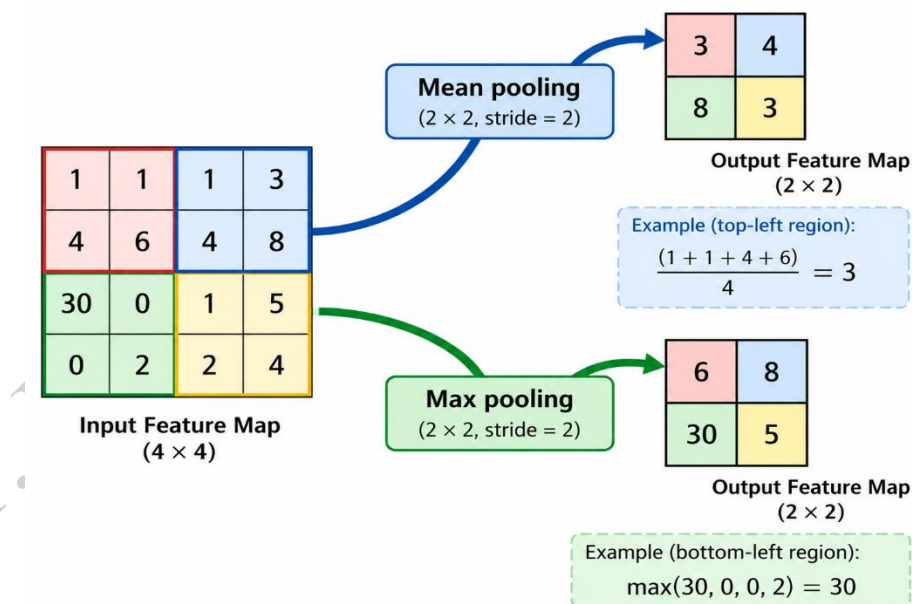


The purpose of apply for the rectifier function is to increase the non-linearity in our images. The **ReLU** function takes a value x and returns 0 if x is negative and x if x is positive. As you can see from the graph, the ReLU function is nonlinear. In this layer, the ReLU function is applied to each point in the feature map. The result is a feature map without negative values.

Pooling layer

The **pooling layer** reduces the size of the image, as it combines neighboring pixels of a certain area of the image into a single representative value. The neighboring pixels are usually selected from the square matrix, and the number of pixels that are combined differs from problem to problem. The representative value is usually set as the *mean* or *maximum* of the selected pixels. The **pooling operation** sweeps a filter across the entire input, but the difference is that this filter does not have any weights.

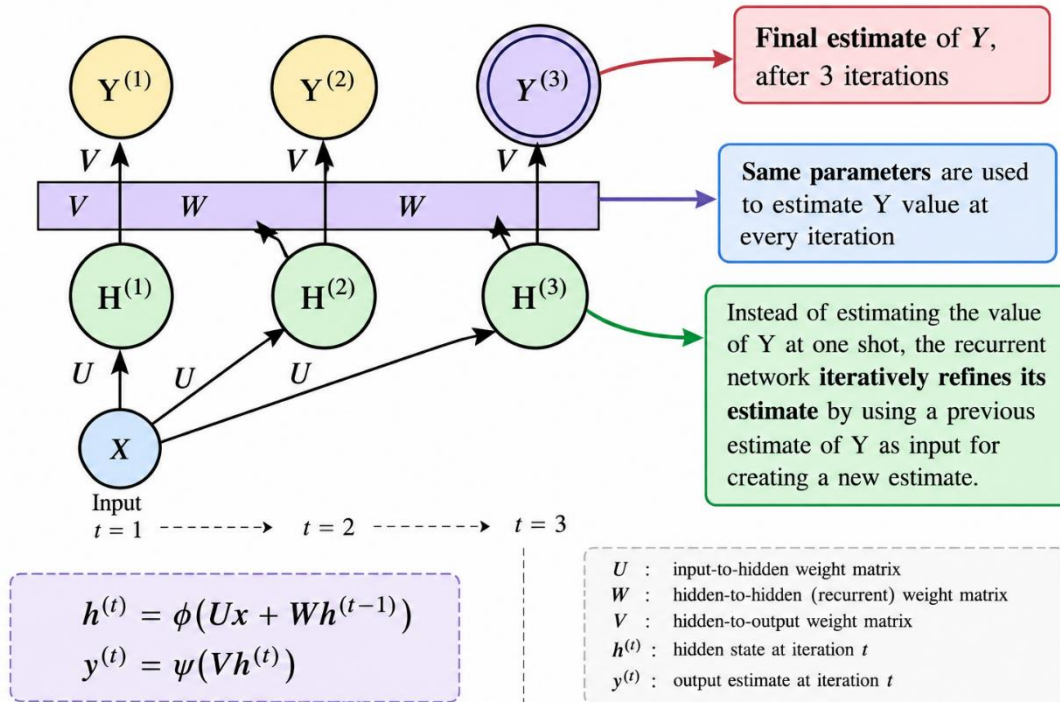
Consider the 4×4 -pixel input image, we combine the pixels of the input image into a 2×2 matrix without overlapping the elements. Once the input image passes through the pooling layer, it shrinks into a 2×2 -pixel image. Figure below shows the resultant cases of pooling using the mean pooling and max pooling.



For example, the pooling layer can improve the recognition of a cat, which may be off-center in the input image. In addition, as the pooling process reduces the image size, it is highly beneficial for relieving the computational load and preventing overfitting.

Fully connected layer or dense layer

Fully connected means that *every output that's produced at the end of the last pooling layer as an input to each node in this fully connected layer*. Neurons in a fully connected layer have full connections to all the activations in the previous layer. The role of the **fully connected layer** is to produce a list of class scores and perform classification based on image features that have been extracted by the previous convolutional and pooling layers. So, the last fully connected layer will have as many nodes as there are classes.



The input layer receives raw data, such as an image, and serves as the starting point for processing. Then the so-called **Hidden Layers**: here the Hidden Layers are *Convolutional Layer, Activation Function, and Pooling Layer*.

Softmax layer

Softmax converts class scores into normalized class probabilities:

$$P(k) = \exp(y(k)) / \sum(j) \exp(y(j))$$

Each $P(k)$ lies between 0 and 1, and the class probabilities sum to 1. The predicted class is commonly the class with the maximum softmax probability. MATLAB provides softmax Layer as a built-in layer

Procedure:

The following architecture is suitable as a methodological example for multi-class image classification.

1. Load the built-in digit dataset from the MATLAB using these commands:

```
[XTrain, YTrain] = digitTrain4DArrayData;  
[XTest, YTest] = digitTest4DArrayData;
```

2. Plotting the training and testing images:

```
% For training images:
```

```
figure;
```

```
for i = 1:16
```

```
    subplot(4,4,i);
```

```
    imshow(XTrain(:,:,1,i));
```

```
    title('Digit: ' + string(YTrain(i)));
```

```
end
```

```
% For Testing images:
```

```
figure;
```

```
for i = 1:16
```

```
    subplot(4,4,i);
```

```
    imshow(XTest(:,:,1,i));
```

```
    title('Digit: ' + string(YTest(i)));
```

```
end
```

3. Constructing the CNN layers:

```
layers = [
```

```
    imageInputLayer([28 28 1])
```

```
    convolution2dLayer(3,8,'Padding','same') %filter size is 3 × 3.
```

```
    batchNormalizationLayer % normalizes
```

```
    reluLayer
```

```
    maxPooling2dLayer(2,'Stride',2)
```

```
    % This layer reduces the size of the feature maps.
```

```
    % For each 2 × 2 region, MATLAB keeps only the largest value.
```

```
    fullyConnectedLayer(10)
```

```
    softmaxLayer
```

```
    classificationLayer];
```

4. Constructing Training Options:

```
options = trainingOptions('sgdm', ...
```

```
    'MaxEpochs',5, ...
```

```
    'MiniBatchSize',64,...
```

```
    'Plots','training-progress');
```

5. Train model

```
net = trainNetwork(XTrain,YTrain,layers,options);
```

6. Test model

```
YPred = classify(net,XTest);
```

```
accuracy = mean(YPred == YTest);
```

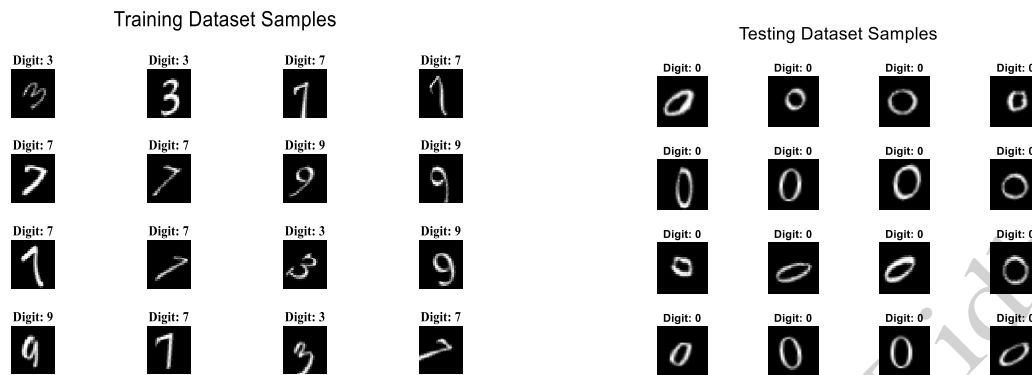
```
fprintf('Accuracy = %.2f%%\n',accuracy*100);
```

7. Plotting results

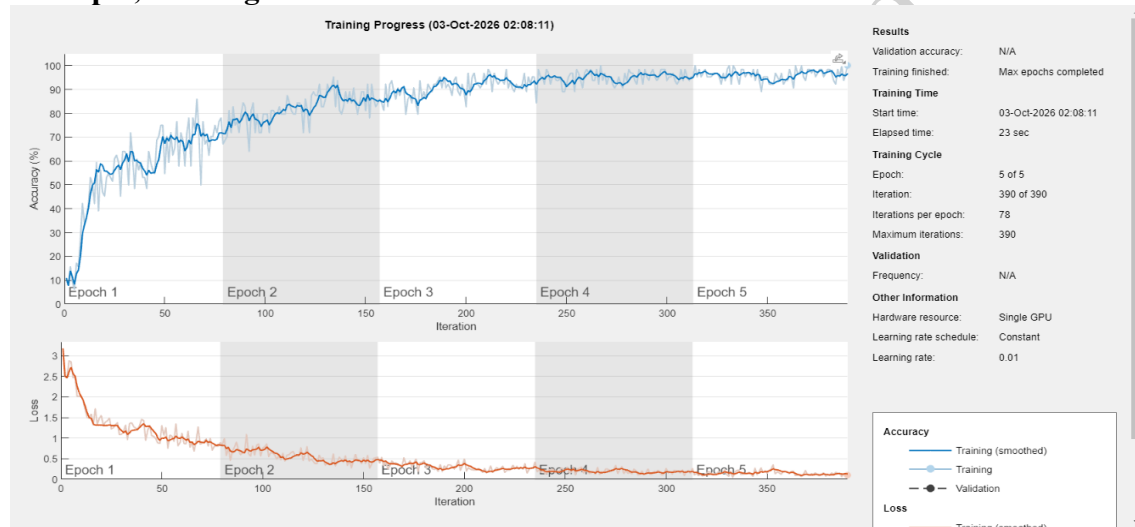
```
figure;  
confusionchart(YTest,YPred);  
title('CNN Classification Results');  
8. Showing the Convolution Filters  
weights = net.Layers(2).Weights;  
figure;  
for i = 1:size(weights,4)  
    subplot(2,4,i);  
    imagesc(weights(:, :, 1, i));  
    axis image off;  
    title('Filter ' + i);  
end  
9. Display the predicted final output  
figure;  
idx = randperm(size(XTest,4),16);  
for i = 1:16  
    subplot(4,4,i);  
    imshow(XTest(:, :, 1, idx(i)));  
    trueLabel = string(YTest(idx(i)));  
    predLabel = string(YPred(idx(i)));  
    title('True: ' + trueLabel + ' Pred: ' + predLabel);  
end
```

Results:

1. Step 2, showing the loaded images of training and testing



2. Step 5, training the CNN



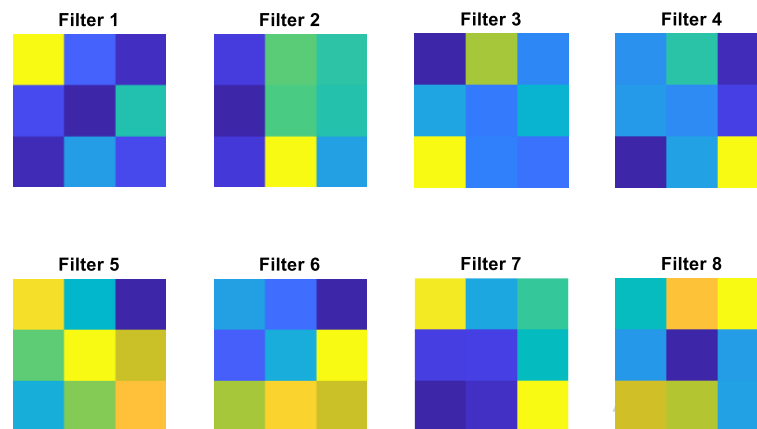
3. Step 7, CNN classification results

CNN Classification Results

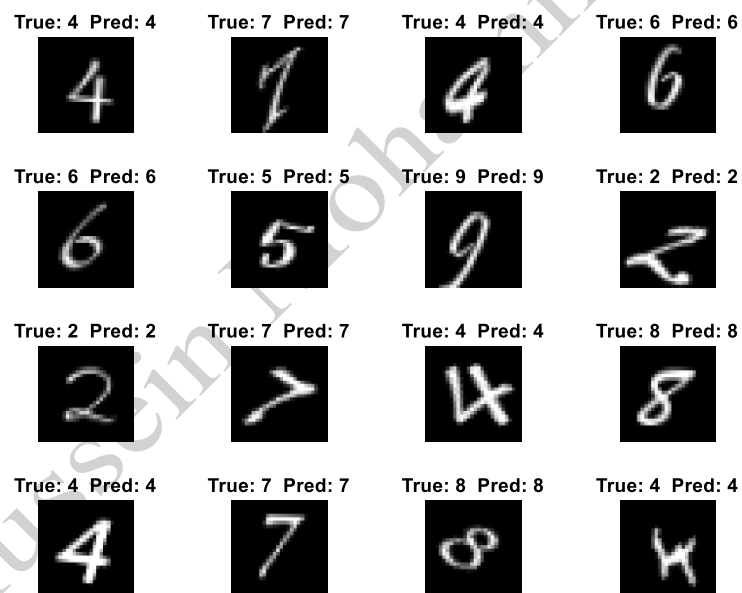
True Class \ Predicted Class	0	1	2	3	4	5	6	7	8	9
0	473	1	4	1	1		4		6	10
1	2	480	7	2	3			4	2	
2			484	3			3	2	3	5
3			4	461	2	3	1		15	14
4		6	1	1	480		1		10	1
5				22		451	8	1	13	5
6	2	8	2	4	3	1	442		30	8
7	2	3	2					493		
8		2	3	2	1		1	2	486	3
9		2		3	1		1	6	22	465

4. Step 8, Convolutional Filter

Learned Convolution Filters



5. Step 9, final predicted output





Dr. Hussein Mohammed Ridha