October 24, 2004
Prepared by: Murad Abu-Khalaf

EE 5322 Neural Networks Notes

This short note on neural networks is based on [1], [2]. Much of this note is based almost entirely on examples and figures taken from these two sources. The MATLAB Neural Networks Toolbox 4.0, [1], is capable of implementing all the learning algorithms that will be presented here. For further readings on this subject, a beginner would find it very beneficial to start with the work of Hagan et al., [2]. A more rigorous take on this subject is by pursuing the work of Haykin, [4]. Both textbooks introduce this subject for the general audience focusing on a broad range of topics from pattern recognition, system identification, filtering theory, data classification, clustering, filtering theory, etc. In the work of Lewis et al., [4], neural networks are rigorously used for adaptive control of robotics and nonlinear systems.

1. Artificial Neural Networks Architectures

Artificial Neural networks are mathematical entities that are modeled after existing biological neurons found in the brain. All the mathematical models are based on the basic block known as artificial neuron. A simple neuron is shown in figure 1. This is 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 **Wp**, the dot product of the (single row) matrix **W** and the vector **p**.

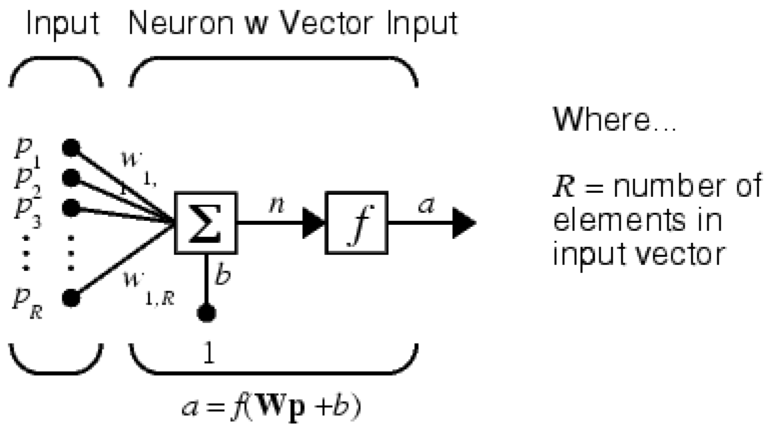


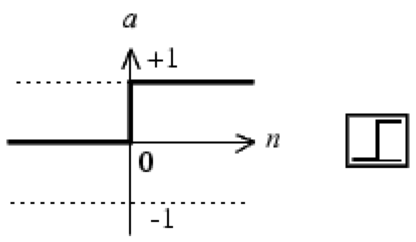
Fig. 1. Simple neuron.

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*

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

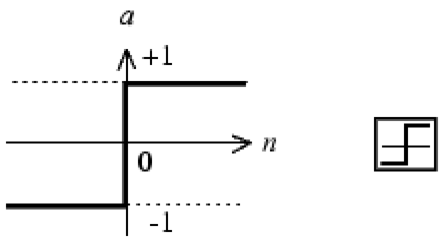
The transfer function *f* can be one of the following commonly used functions in neural network literature:

October 24, 2004
Prepared by: Murad Abu-Khalaf



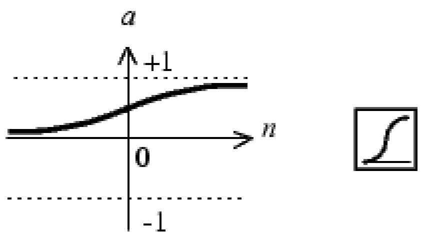
$a = hardlim(n)$

Hard-Limit Transfer Function



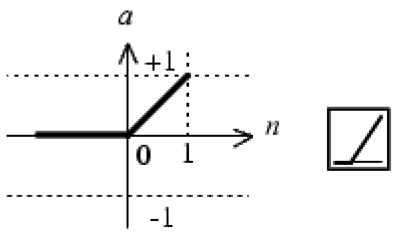
$a = hardlims(n)$

Symmetric Hard-Limit Trans. Funct.



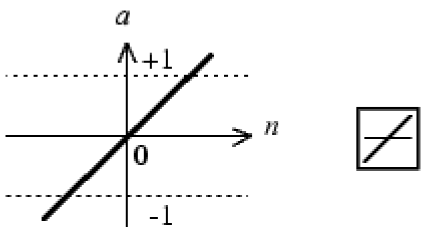
$a = logsig(n)$

Log-Sigmoid Transfer Function



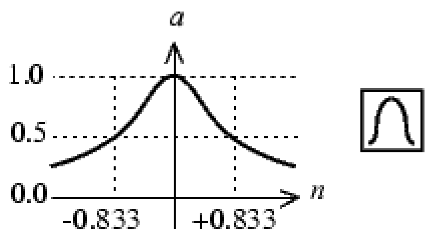
$a = poslin(n)$

Positive Linear Transfer Funct.



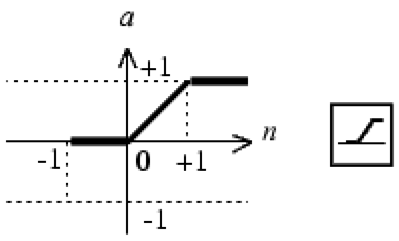
$a = purelin(n)$

Linear Transfer Function



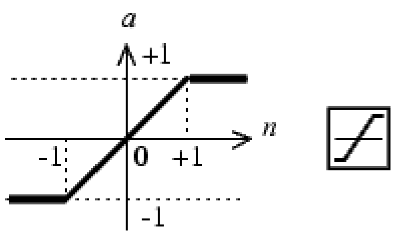
$a = radbas(n)$

Radial Basis Function



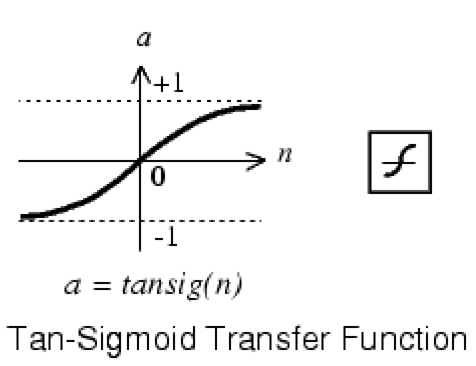
$a = satlin(n)$

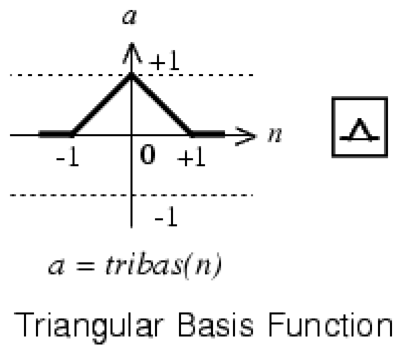
Satlin Transfer Function



$a = satlins(n)$

Satlins Transfer Function





Introducing vector notations, figure 1 can be rewritten in a more compact representation as seen in figure 2.

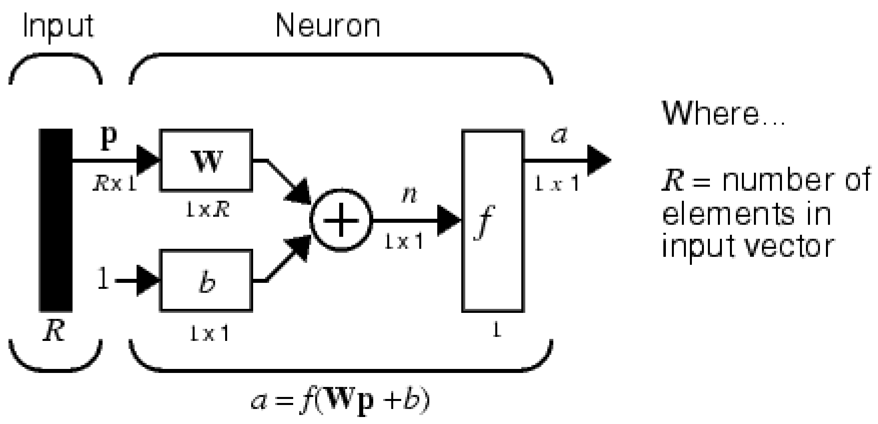
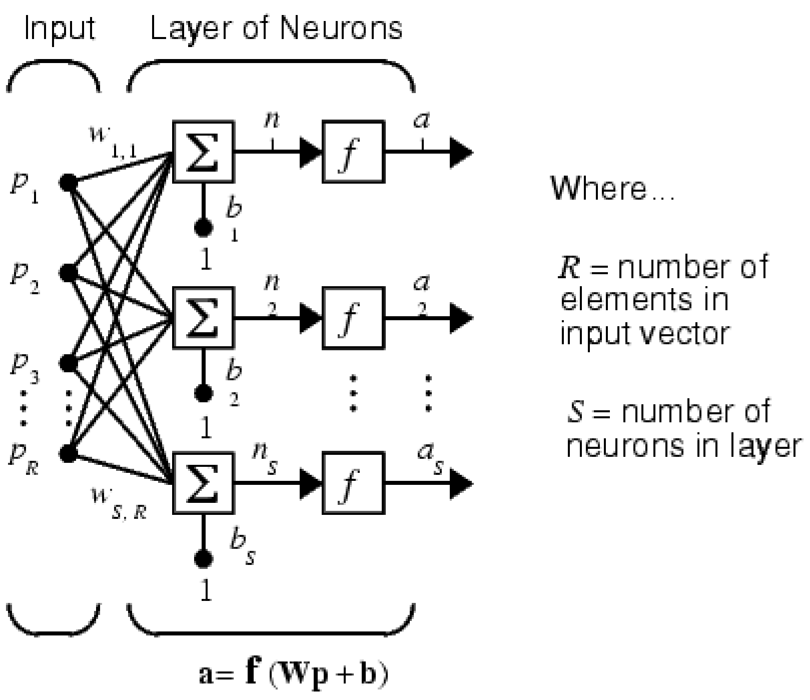

Input
Neuron
p
Rx1
W
1xR
1
b
1x1
n
1x1
f
a
1x1
1
R
$a = f(\mathbf{W}\mathbf{p} + b)$
Where...
R = number of elements in input vector

Fig. 2. Simple neuron in vector notation.

Here the input vector **p** is represented by the solid dark vertical bar at the left. The dimensions of **p** are shown below the symbol **p** in the figure as *R*x1. (Note that we will use a capital letter, such as *R* in the previous sentence, when referring to the *size* of a vector.) Thus, p is a vector of *R* input elements. These inputs post multiply the single row, *R* column matrix **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 **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 one-layer network with *R* input elements and *S* neurons is shown in figure 3.



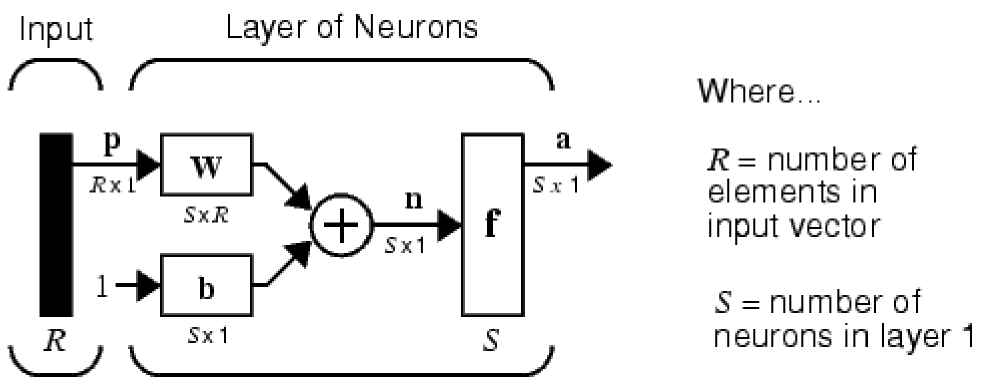
Input
Layer of Neurons
$w_{1,1}$
p_1
p_2
p_3
p_R
$w_{S,R}$
n_1
n_2
n_S
a_1
a_2
a_S
b_1
b_2
b_S
1
f
Where...
R = number of elements in input vector
S = number of neurons in layer
$\mathbf{a} = \mathbf{f}(\mathbf{Wp} + \mathbf{b})$

Fig. 3. One-layer network.

In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **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 **n**. Finally, the neuron layer outputs form a column vector **a**. We show the expression for **a** at the bottom of the figure.
Note that 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.
You can create a single (composite) layer of neurons having different transfer functions 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 **W**.

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

The *S* neuron *R* input one-layer network also can be drawn in abbreviated notation.

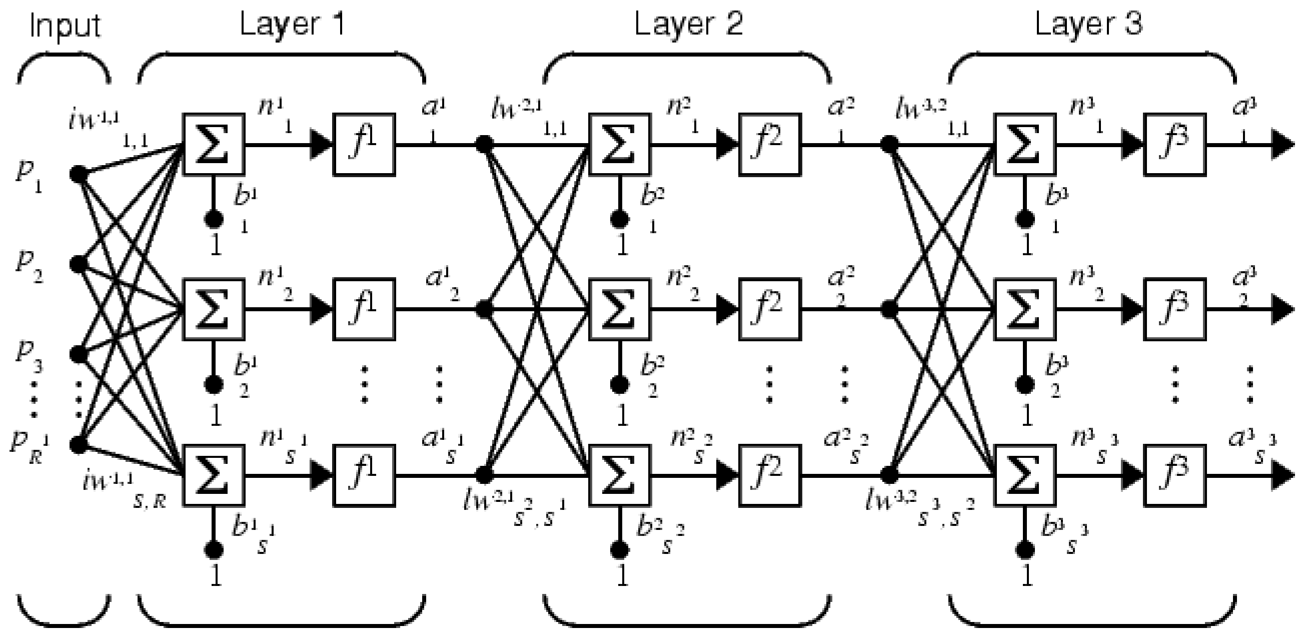

Input
Layer of Neurons
Where...
p
R x 1
W
S x R
1
b
S x 1
n
S x 1
f
a
S x 1
R
S
R = number of elements in input vector
S = number of neurons in layer 1

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

Fig. 4. One-layer network in vector notation.

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

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.

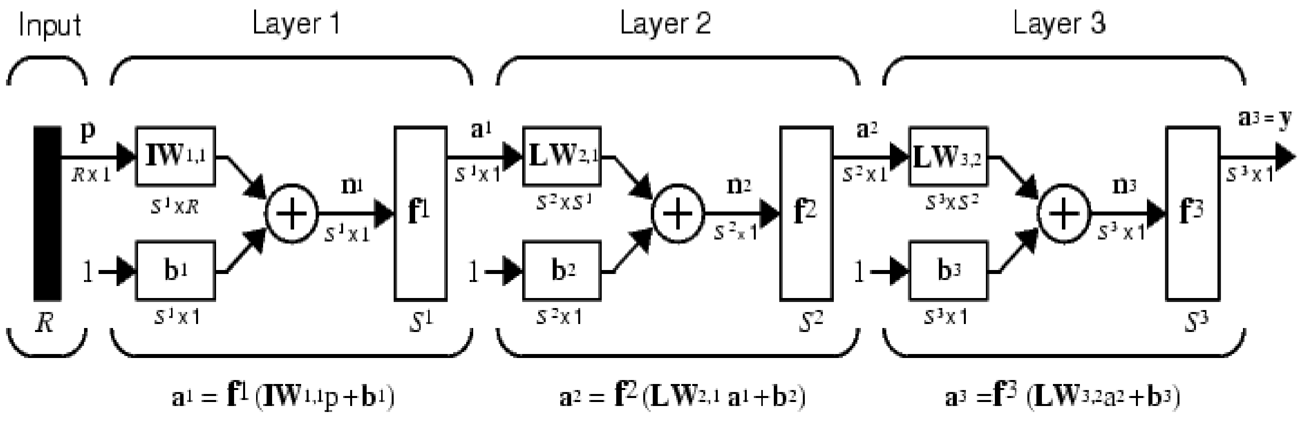


$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1 + \mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2 + \mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

Fig. 5. Multi-layer network.

The network shown above 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 numbers of neurons. A constant input 1 is fed to the biases for each neuron.
Note that 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^2xS^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. This approach can be taken with any layer of the network.
The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. We will not use that designation.
The same three-layer network discussed previously also can be drawn using our abbreviated notation.



$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3 = \mathbf{y}$$

Fig. 6. Multi-layer network in vector 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. Here we assume that the output of the third layer, a^3, is the network output of interest, and we have labeled this output as y. We will use this notation to specify the output of multilayer networks.

The previous networks considered are ***Feedforward*** in the sense of the flow of information through the network. There exist neural network architectures in which the flow of information can have loops. These are called ***Recurrent*** networks, i.e. Hopfield networks seen in the following figure.

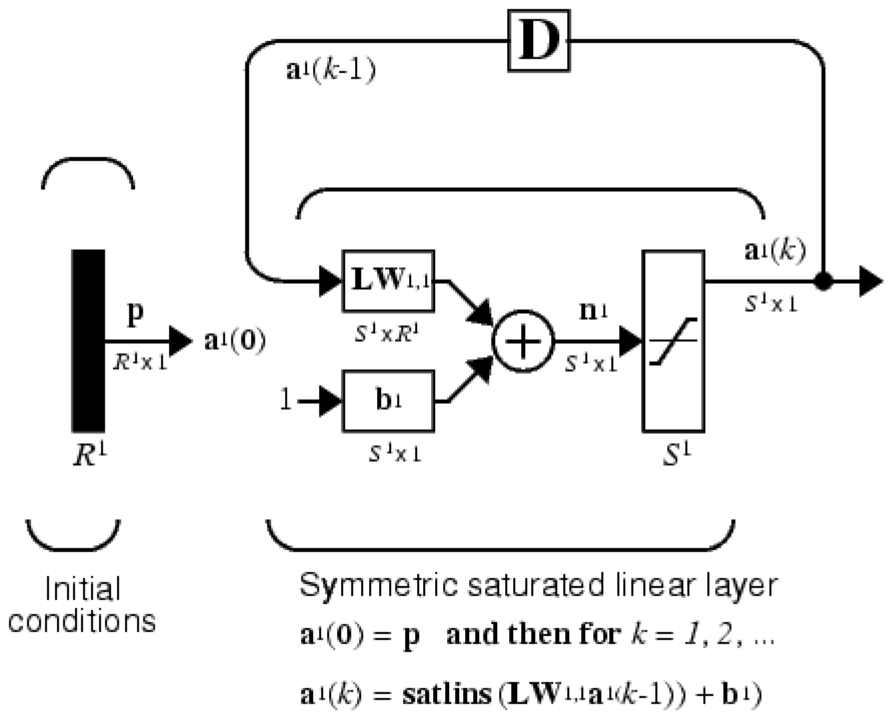


Fig. 7. Recurrent neural network.

The *input* **p** to this network merely supplies the initial conditions.

2. Learning and Training Neural Networks

Training of neural networks is done by devising learning rules. A learning rule is 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 particular task. Learning can be categorized into:

Supervised learning: the learning rule is provided with a set of examples (the *training set*) of proper network behavior

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

where $\mathbf{p_q}$ is an input to the network, and $\mathbf{t_q}$ 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 in order to move the network outputs closer to the targets. The *Perceptron learning* rule falls in this supervised learning category. There is also the *Supervised Hebbian learning.*

Supervised learning is used in general to tackle pattern recognition, data classification, and function approximation problems.

Unsupervised learning: the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization. *Unsupervised Hebbian learning, Competitive learning*, and *Associative learning* are examples of this.

Reinforcement Learning: Similar to supervised learning, except that, instead of providing the correct output for each network, the algorithm is only a given a grade which indicates the performance of the network. Think of it as Reward/Penalty type of learning. *Temporal difference learning*, *Q-learning*, *Value-Iteration*, *Policy Iterations*, *Adaptive Critics*, are all variants of reinforcement learning methods.

Reinforcement learning is used extensively to solve problems involving optimal control, Markov Decision Problems MDP, and other dynamics programming related learning problems.

In this short document, we focus on supervised learning. In particular, we briefly mention Perception learning, Supervised Hebbian learning, LMS, and Error Backpropagation. It is recommended though to learn this material from the referenced sources [1], [2].

A) Perceptron Learning Rule

The perceptron learning rule is used for single-layer perceptron networks training. Figure 8 shows a perceptron network. The earliest kind of neural network is a *single-layer perceptron* network, which consists of a single layer of output nodes; the inputs are fed directly to the outputs via a series of weights. In this way it can be considered the simplest kind of feedforward network.

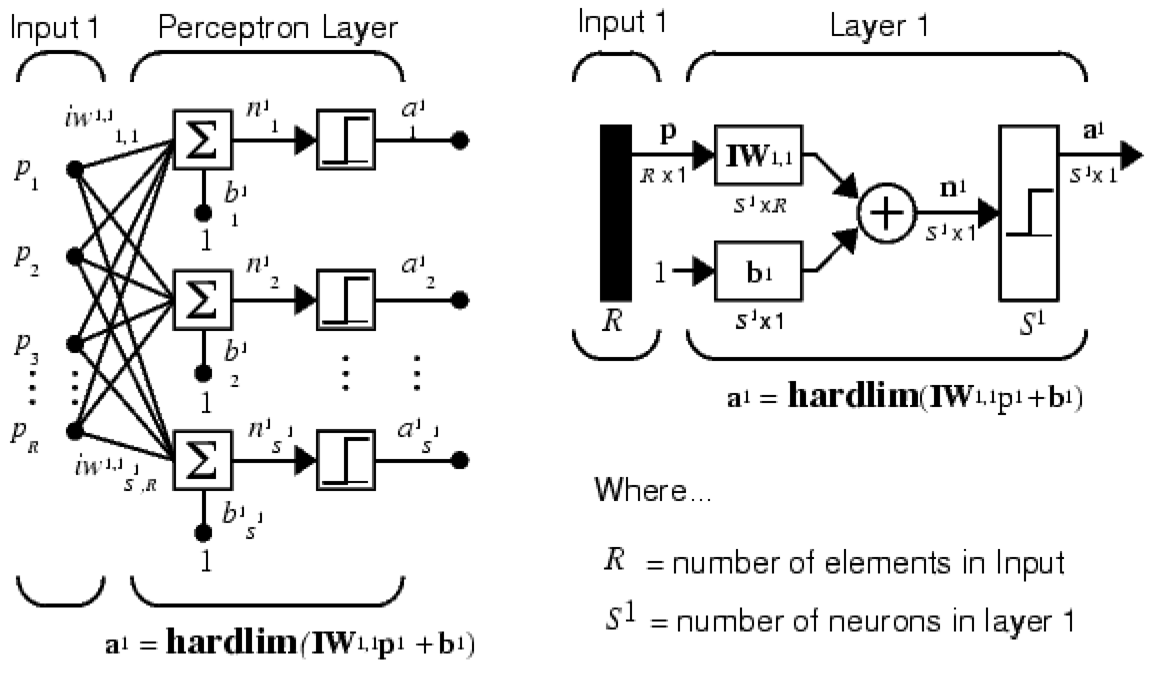


Fig. 8. Single-Layer Perceptron Network.

In a perceptron network, each neuron divides the input space into two regions. Boundaries between regions are called decision boundaries. For more information, consult a print out of [1], and the book [2].

The perceptron training rule is given by

$$W^{new} = W^{old} + ep^T,$$
$$b^{new} = b^{old} + e.$$

where **e**=**t-a.**

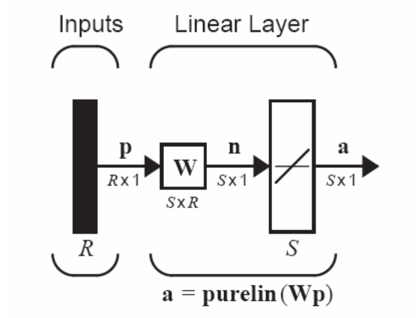
Perceptron learning will converge if the problem at hand is linearly separable.

B) Supervised Hebbian Learning

This supervised Hebbian learning rule can be simply stated as follows

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}$$

We can test the Hebbian learning rule on a linear associator shown in the following figure.



$$\mathbf{a} = \mathbf{Wp} \qquad a_i = \sum_{j=1}^{R} w_{ij} p_j$$

The Hebbian learning rule becomes

$$\mathbf{W} = \mathbf{t}_1\mathbf{p}_1^T + \mathbf{t}_2\mathbf{p}_2^T + \cdots + \mathbf{t}_Q\mathbf{p}_Q^T = \sum_{q=1}^{Q} \mathbf{t}_q\mathbf{p}_q^T$$

$$\mathbf{W} = \begin{bmatrix}\mathbf{t}_1 & \mathbf{t}_2 & \ldots & \mathbf{t}_Q\end{bmatrix}\begin{bmatrix}\mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T\end{bmatrix} = \mathbf{TP}^T \qquad \mathbf{P} = \begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_2 & \ldots & \mathbf{p}_Q\end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix}\mathbf{t}_1 & \mathbf{t}_2 & \ldots & \mathbf{t}_Q\end{bmatrix}$$

Now to test the output of the net after training, $a = WP = TP^TP$. If the neural network did learn correctly, then $a = WP = TP^TP = T$. Which implies that only when P^TP, that is orthonormal matrix P, we have perfect learning.

In general, and for networks not necessary linear associator, the learning rule $W = TP^T$ should be replaced by $W = TP^+$, where P^+ is the pseudoinverse, and $W = TP^+$ is the least squares solution of the error $\|E\|^2 = \|T - WP\|^2$.

C) Widrow-Hopf Learning – (LMS Algorithm)

Like the perceptron learning rule, 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:

$$\{\mathbf{p}_1,\mathbf{t}_1\},\{\mathbf{p}_2,\mathbf{t}_2\},\ldots,\{\mathbf{p}_Q,\mathbf{t}_Q\}$$

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 of the 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 linear network so as to minimize this mean square error. The key insight of Widrow and Hopf was that they could estimate the mean squared error by

$$m\hat{s}e = (e(k))^2 = (t(k)-a(k))^2 .$$

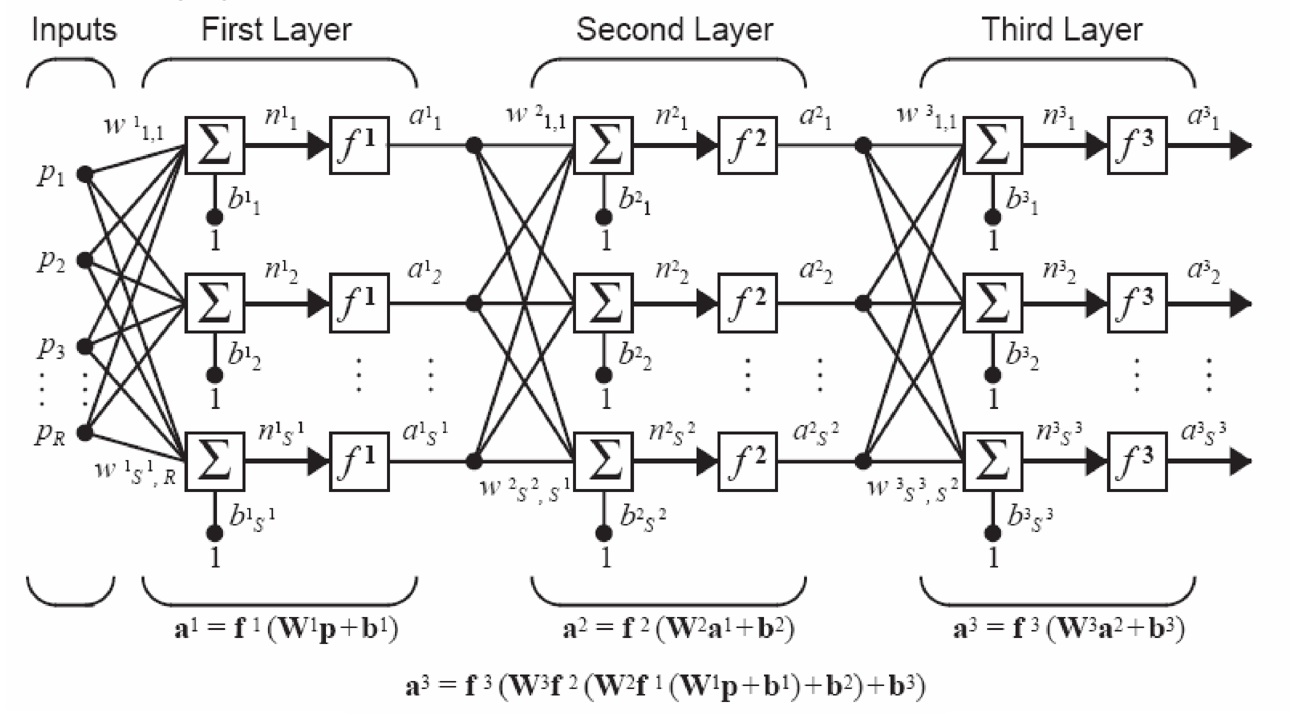
For a complete derivation of the LMS algorithm, refer to the overhead transparencies or [2].

The algorithm is finally given by

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

D) Error Backpropagation

The aim here is to use the LMS algorithm to train a neural network that has multilayers as shown in the following figure



$R - S^1 - S^2 - S^3$ Network

For a complete derivation of the backpropagation algorithm, refer to the overhead transparencies or [2].

The Backpropagation algorithm works as follows.

First: *Forward Propagation:*

Here, given some **p,** we calculate the final out of the network by propagating forward as follows:

$$\mathbf{a}^0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \qquad m = 0, 2, \ldots, M-1$$

$$\mathbf{a} = \mathbf{a}^M$$

Second: *Backpropagation:*

In this step, we adjust the weights and biases by implementing the LMS algorithm. The LMS algorithm for this case has access to the error signal at the output layer only, and therefore, errors are back propagated from the output layer using the so called sensitivity functions. The sensitivity function at the output layer is given by

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a})$$

For all other layers, we use the following backward difference equation

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1} \qquad m = M-1, \ldots, 2, 1$$

After the sesetivity has been deterimined fo each neuron, weight update can take place as follows

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T$$
$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m$$

E) Hopfield networks design

The Hopfield network may be viewed as a nonlinear *associative memory* or *content addressable memory*. In an associative memory the desired output vector is equal to the input vector. The network stores certain desired patterns. When a distorted pattern is presented to the network, then it is associated with another pattern. If the network works properly, this associated pattern is one of the stored patterns. In some cases (when the different class patterns are correlated), spurious minima can also appear. This means that some patterns are associated with patterns that are not among the stored patterns.

The Hopfield network can be described as a dynamic system whose phase plane contains a set of fixed (stable) points representing the fundamental memories of the system. Hence the network can help retrieve information and cope with errors. Figure 9 shows a Hopfield network, and figure 10 shows the system theoretic representation of it.

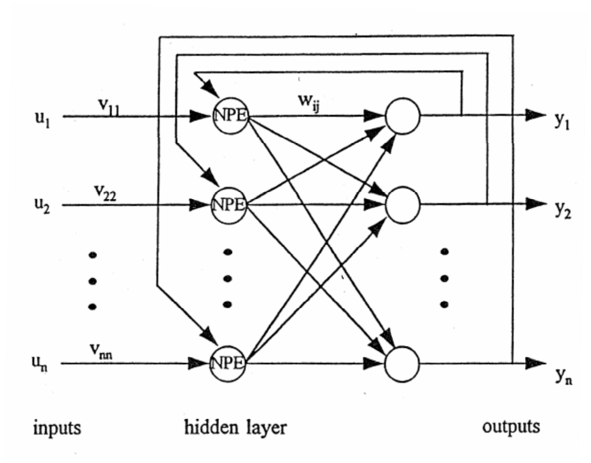


Figure 1.1.11: Hopfield dynamical neural net.

Fig. 9. Hopfield Network.

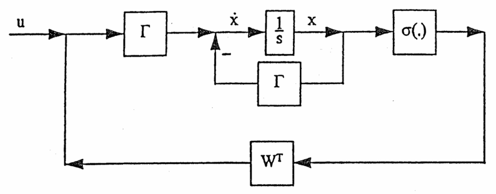


Figure 1.1.14: Continuous-time Hopfield net in block diagram form.

$$\dot{x} = -\Gamma x + \Gamma W^T \sigma(x) + \Gamma u.$$

Fig. 10. System Theoretic Representation of Hopfield Network.

The Hopfield net does not have an iterative learning law as in the perceptron learning rule, etc. a design procedure based on LaSalle's extension of Lyapunov theory. For more on this, consult the text books [2], [3], [4].

In short, the weights of a Hopfield network can be selected as follows

$$W = \frac{1}{n}\sum_{q=1}^{Q} p_q p_q^T$$

where n is the dimension of the pattern vector , and Q is the number of patterns stored.

We can verify that this design law works for the case for which the input patterns are orthogonal. To see this, note that $\dot{x} = -\Gamma x + \Gamma W^T \sigma(x) + \Gamma u$. The activation function, or transfer function, of the net is shown to be *tansig* function. If the weight selection rule is plugged into the dynamics of the system, we have

$$\dot{x} = -\Gamma x + \Gamma\left(\sum_{q=1}^{Q} p_q p_q^T\right)\sigma(x) + \Gamma u \,.$$

Now for any stored pattern p_i and under the assumption that these patterns are orthogonal, the dynamics will have an equilibrium point.

$$\begin{aligned}
\dot{x} &= -\Gamma p_i + \Gamma\left(\frac{1}{n}\sum_{q=1}^{Q} p_q p_q^T\right)\sigma(p_i) \\
&= -\Gamma p_i + \Gamma\left(\frac{1}{n}\sum_{q=1}^{Q} p_q p_q^T\right)p_i \\
&= -\Gamma p_i + \Gamma\left(\frac{1}{n}\sum_{q\neq i} p_q p_q^T p_i + \frac{1}{n} p_i p_i^T p_i\right) \\
&= -\Gamma p_i + \Gamma\left(0 + \frac{1}{n} p_i \cdot n\right) \\
&= 0.
\end{aligned}$$

Note that Hopfield used a Lyapunov function to show that these equilibrium points are stable. Furthermore, these are not the only equilibrium points. Several more undesired equilibrium points appear using this weight selection method. These are called spurious attractors. This means that some patterns are associated with patterns that are not among the stored pattern vectors.

For the purpose of this class, you only need to understand how to use [1] which will automatically do the design of the net for you given the desired patterns.

References:

[1] Demuth, H., M. Beale, *MATLAB Neural Network Toolbox v. 4.0.4*,
http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/
http://www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf
[2] Hagan, M., H. Demuth, M. Beale, *Neural Network Design*, PWS Publishing Company, 1996.
[3] Lewis, F., S. Jagannathan, A. Yesildirek: *Neural Network Control of Robot Manipulators and Nonlinear Systems*, Taylor & Francis, 1999.
[4] Haykin, S., *Neural Networks: A Comprehensive Foundation*, Prentice Hall; 2nd edition, 1998.