Introduction to Neural Networks

Dr. M. Syed Ali

Department of Mathematics

Thiruvalluvar Univeristy

Vellore.

Introduction to Neural Networks

What are Artificial Neural Networks?

Artificial Neural Networks are relatively crude electronic models based on the neural structure of the brain. The brain basically learns from experience. It is natural proof that some problems that are beyond the scope of current computers are indeed solvable by small energy efficient packages. This brain modeling also promises a less technical way to develop machine solutions. This new approach for computing also provides a more graceful degradation during system overload than its more traditional counterparts.

These biologically inspired methods of computing are thought to be the next major advancement in the computing industry. Even simple animal brains are capable of functions that are currently impossible for computers. Computers do rote things well, like keeping ledgers or performing complex math. But computers have trouble recognizing even simple patterns much less generalizing those patterns of the past into actions of the future.

Now, advances in biological research promise an initial understanding of the natural thinking mechanism. This research shows that brains store information as patterns. Some of these patterns are very complicated and allow us the ability to recognize individual faces from many different angles. This process of storing information as patterns, utilizing those patterns, and then solving problems encompasses a new field in computing. This field, as mentioned before, does not utilize traditional programming but involves the creation of massively parallel networks and the training of those networks to solve specific problems. This field also utilizes words very different from traditional computing, words like behave, react, self-organize, learn, generalize, and forget.

Analogy to the Brain

The exact workings of the human brain are still a mystery. Yet, some aspects of this amazing processor are known. In particular, the most basic element of the human brain is a specific type of cell which, unlike the rest of the body, doesn't appear to regenerate. Because this type of cell is the only part of the body that isn't slowly replaced, it is assumed that these cells are what provides us with our abilities to remember, think, and apply previous experiences to our every action. These cells, all 100 billion of them, are known as neurons. Each of these neurons can connect with up to 200,000 other neurons, although 1,000 to 10,000 is typical. The power of the human mind comes from the sheer numbers of these basic components and the multiple connections between them. It also comes from genetic programming and learning.

The individual neurons are complicated. They have a myriad of parts, sub-systems, and control mechanisms. They convey information via a host of electrochemical pathways. There are over one hundred different classes of neurons, depending on the classification method used. Together these neurons and their connections form a process which is not binary, not stable, and not synchronous. In short, it is nothing like the currently available electronic computers, or even artificial neural networks.

These artificial neural networks try to replicate only the most basic elements of this complicated, versatile, and powerful organism. They do it in a primitive way. But for the software engineer who is trying to solve problems, neural computing was never about replicating human brains. It is about machines and a new way to solve problems.

Artificial Neurons and How They Work

The fundamental processing element of a neural network is a neuron. This building block of human awareness encompasses a few general capabilities.

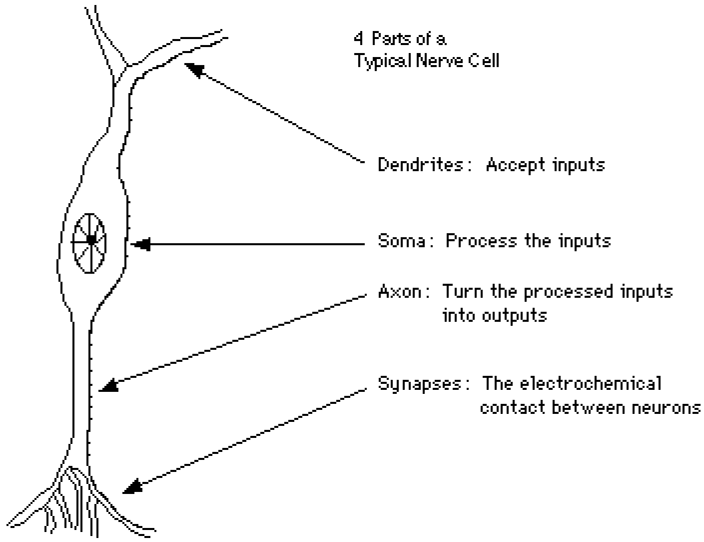


Figure 1: A Simple Neuron.

Basically, a biological neuron receives inputs from other sources, combines them in some way, performs a generally nonlinear operation on the result, and then outputs the final result. Figure 1 shows the relationship of these four parts.

Within humans there are many variations on this basic type of neuron, further complicating man's attempts at electrically replicating the process of thinking. Yet, all natural neurons have the same four basic components. These components are known by their biological names - dendrites, soma, axon, and synapses. Dendrites are hair-like extensions of the soma which act like input channels. These input channels receive their input through the synapses of other neurons. The soma then processes these incoming signals over time. The soma then turns that processed value into an output which is sent out to other neurons through the axon and the synapses.

Recent experimental data has provided further evidence that biological neurons are structurally more complex than the simplistic explanation above. They are significantly more complex than the existing artificial neurons that are built into today's artificial neural networks. As biology provides a better understanding of neurons, and as technology advances, network designers can continue to improve their systems by building upon man's understanding of the biological brain.

But currently, the goal of artificial neural networks is not the grandiose recreation of the brain. On the contrary, neural network researchers are seeking an understanding of nature's capabilities for which people can engineer solutions to problems that have not been solved by traditional computing. To do this, the basic unit of neural networks, the artificial neurons, simulate the four basic functions of natural neurons.

Figure 2 shows a fundamental representation of an artificial neuron where various inputs to the network are represented by the mathematical symbol, x(n). Each of these inputs are multiplied by a connection weight. These weights are represented by w(n). In the simplest case, these products are simply summed, fed through a transfer function to generate a result, and then output. This process lends itself to physical implementation on a large scale in a small package. This electronic implementation is still possible with other network structures which utilize different summing functions as well as different transfer functions.

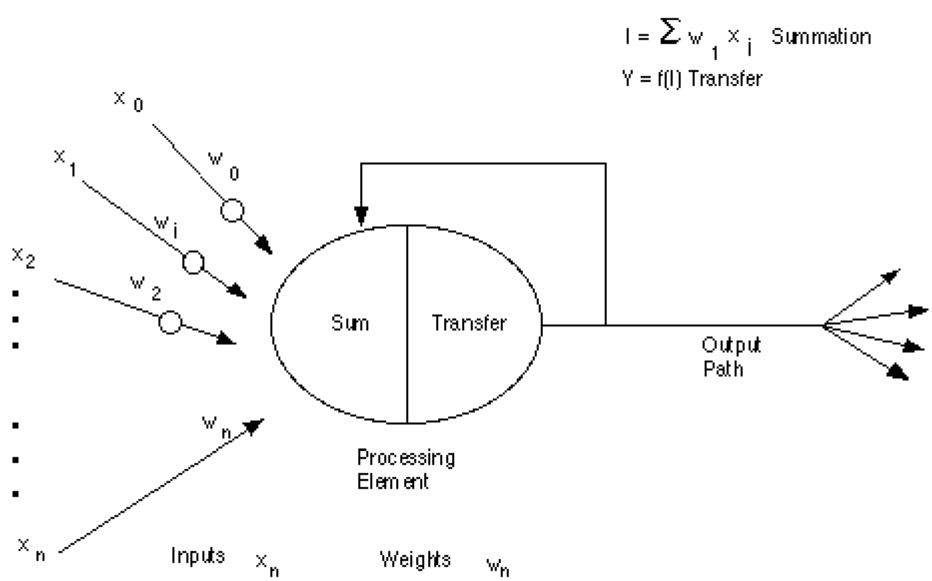


Figure 2: A Basic Artificial Neuron.

Some applications require "black and white," or binary, answers. These applications include the recognition of text, the identification of speech, and the image deciphering of scenes. These applications are required to turn real-world inputs into discrete values. These potential values are limited to some known set, like the ASCII characters or the most common 50,000 English words. Because of this limitation of output options, these applications don't always utilize networks composed of neurons that simply sum up, and thereby smooth, inputs. These networks may utilize the binary properties of ORing and ANDing of inputs. These functions, and many others, can be built into the summation and transfer functions of a network.

Other networks work on problems where the resolutions are not just one of several known values. These networks need to be capable of an infinite number of responses. Applications of this type include the "intelligence" behind robotic movements. This "intelligence" processes inputs and then creates outputs which actually cause some device to move. That movement can span an infinite number of very precise motions. These networks do indeed want to smooth their inputs which, due to limitations of sensors, comes in non-continuous bursts, say thirty times a second. To do that, they might accept these inputs, sum that data, and then produce an output by, for example, applying a hyperbolic tangent as a transfer function. In this manner, output values from the network are continuous and satisfy more real world interfaces.

Other applications might simply sum and compare to a threshold, thereby producing one of two possible outputs, a zero or a one. Other functions scale the outputs to match the application, such as the values minus one and one. Some functions even integrate the input data over time, creating time-dependent networks.

Electronic Implementation of Artificial Neurons

In currently available software packages these artificial neurons are called "processing elements" and have many more capabilities than the simple artificial neuron described above. Those capabilities will be discussed later in this report. Figure 3 is a more detailed schematic of this still simplistic artificial neuron.

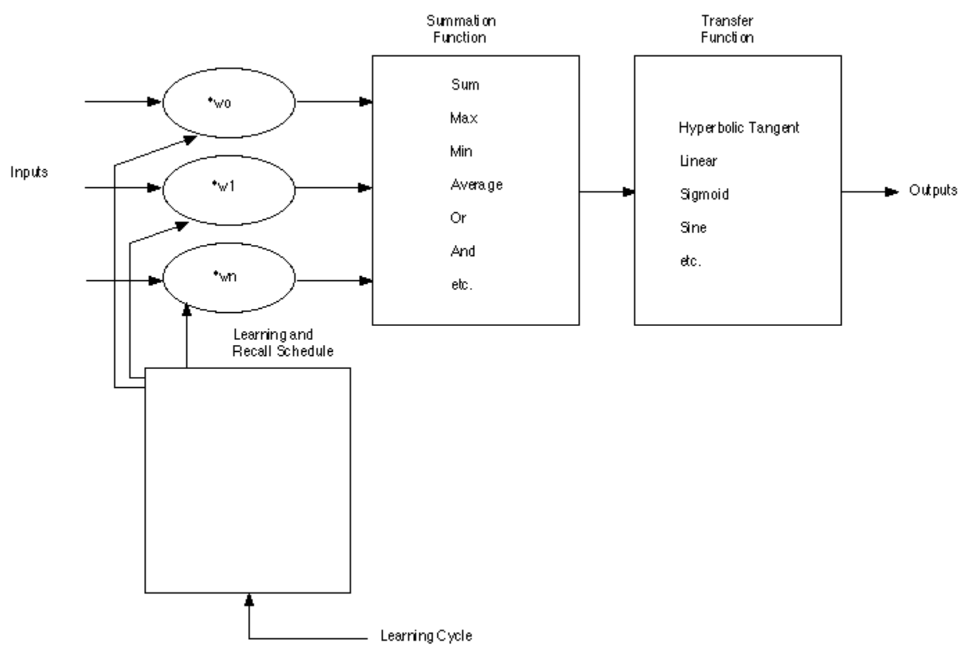


Figure 3: A Model of a "Processing Element".

In Figure 3, inputs enter into the processing element from the upper left. The first step is for each of these inputs to be multiplied by their respective weighting factor (w(n)). Then these modified inputs are fed into the summing function, which usually just sums these products. Yet, many different types of operations can be selected. These operations could produce a number of different values which are then propagated forward; values such as the average, the largest, the smallest, the ORed values, the ANDed values, etc. Furthermore, most commercial development products allow software engineers to create their own summing functions via routines coded in a higher level language (C is commonly supported). Sometimes the summing function is further complicated by the addition of an activation function which enables the summing function to operate in a time sensitive way.

Either way, the output of the summing function is then sent into a transfer function. This function then turns this number into a real output via some algorithm. It is this algorithm that takes the input and turns it into a zero or a one, a minus one or a one, or some other number. The transfer functions that are commonly supported are sigmoid, sine, hyperbolic tangent, etc. This transfer function also can scale the output or control its value via thresholds. The result of the transfer function is usually the direct output of the processing element. An example of how a transfer function works is shown in Figure 4. This sigmoid transfer function takes the value from the summation function, called sum in the Figure 4, and turns it into a value between zero and one.

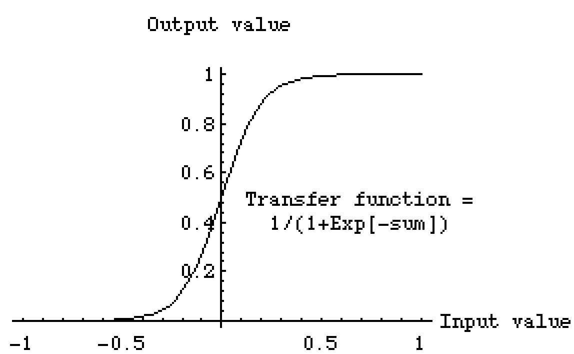


Figure 4: Sigmoid Transfer Function.

Finally, the processing element is ready to output the result of its transfer function. This output is then input into other processing elements, or to an outside connection, as dictated by the structure of the network. All artificial neural networks are constructed from this basic building block - the processing element or the artificial neuron. It is variety and the fundamental differences in these building blocks which partially cause the implementing of neural networks to be an "art."

Artificial Network Operations

The other part of the "art" of using neural networks revolve around the myriad of ways these individual neurons can be clustered together. This clustering occurs in the human mind in such a way that information can be processed in a dynamic, interactive, and self-organizing way. Biologically, neural networks are constructed in a three-dimensional world from microscopic components. These neurons seem capable of nearly unrestricted interconnections. That is not true of any proposed, or existing, man-made network. Integrated circuits, using current technology, are two-dimensional devices with a limited number of layers for interconnection. This physical reality restrains the types, and scope, of artificial neural networks that can be implemented in silicon.

Currently, neural networks are the simple clustering of the primitive artificial neurons. This clustering occurs by creating layers which are then connected to one another. How these layers connect is the other part of the "art" of engineering networks to resolve real world problems.

Basically, all artificial neural networks have a similar structure or topology as shown in Figure 5. In that structure some of the neurons interfaces to the real world to receive its inputs. Other neurons provide the real world with the network's outputs. This output might be the particular character that the network thinks that it has scanned or the particular image it thinks is being viewed. All the rest of the neurons are hidden from view.

But a neural network is more than a bunch of neurons. Some early researchers tried to simply connect neurons in a random manner, without much success. Now, it is known that even the brains of snails are structured devices. One of the easiest ways to design a structure is to create layers of elements. It is the grouping of these neurons into layers, the connections between these layers, and the summation and transfer functions that comprises a functioning neural network. The general terms used to describe these characteristics are common to all networks.

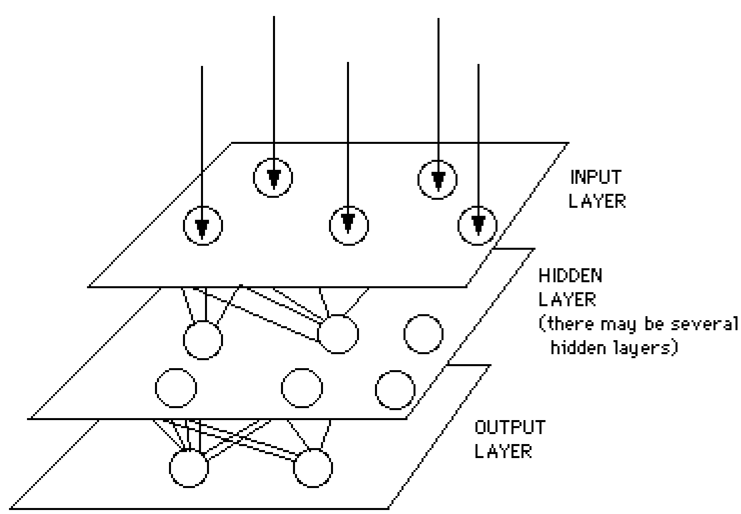


Figure 5: A Simple Neural Network Diagram.

Although there are useful networks which contain only one layer, or even one element, most applications require networks that contain at least the three normal types of layers - input, hidden, and output. The layer of input neurons receives the data either from input files or directly from electronic sensors in real-time applications. The output layer sends information directly to the outside world, to a secondary computer process, or to other devices such as a mechanical control system. Between these two layers can be many hidden layers. These internal layers contain many of the neurons in various interconnected structures. The inputs and outputs of each of these hidden neurons simply go to other neurons.

In most networks each neuron in a hidden layer receives the signals from all of the neurons in a layer above it, typically an input layer. After a neuron performs its function it passes its output to all of the neurons in the layer below it, providing a feedforward path to the output.

These lines of communication from one neuron to another are important aspects of neural networks. They are the glue to the system. They are the connections which provide a variable strength to an input. There are two types of these connections. One causes the summing mechanism of the next neuron to add while the other causes it to subtract. In more human terms one excites while the other inhibits.

Some networks want a neuron to inhibit the other neurons in the same layer. This is called lateral inhibition. The most common use of this is in the output layer. For example in text recognition if the probability of a character being a "P" is .85 and the probability of the character being an "F" is .65, the network wants to choose the highest probability and inhibit all the others. It can do that with lateral inhibition. This concept is also called competition.

Another type of connection is feedback. This is where the output of one layer routes back to a previous layer. An example of this is shown in Figure 6.

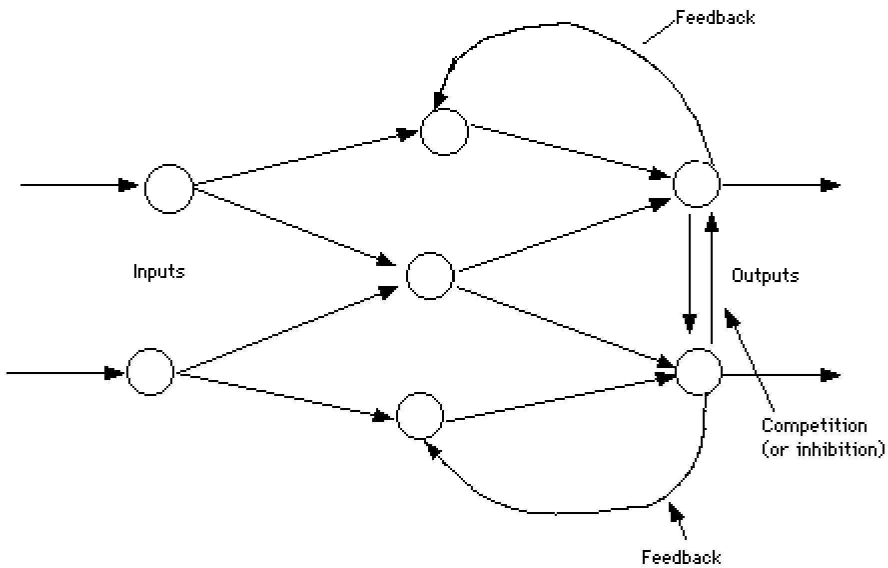


Figure 6: Simple Network with Feedback and Competition.

The Biological Inspiration

Neural networks grew out of research in Artificial Intelligence; specifically, attempts to mimic the fault-tolerance and capacity to learn of biological neural systems by modeling the low-level structure of the brain. The main branch of Artificial Intelligence research in the 1960s -1980s produced Expert Systems. These are based upon a high-level model of reasoning processes (specifically, the concept that our reasoning processes are built upon manipulation of symbols). It became rapidly apparent that these systems, although very useful in some domains, failed to capture certain key aspects of human intelligence. According to one line of speculation, this was due to their failure to mimic the underlying structure of the brain. In order to reproduce intelligence, it would be necessary to build systems with a similar architecture.

The brain is principally composed of a very large number (circa 10,000,000,000) of neurons, massively interconnected (with an average of several thousand interconnects per neuron, although this varies enormously). Each neuron is a specialized cell which can propagate an electrochemical signal. The neuron has a branching input structure (the dendrites), a cell body, and a branching output structure (the axon). The axons of one cell connect to the dendrites of another via a synapse. When a neuron is activated, it fires an electrochemical signal along the axon. This signal crosses the synapses to other neurons, which may in turn fire. A neuron fires only if the total signal received at the cell body from the dendrites exceeds a certain level (the firing threshold).

The strength of the signal received by a neuron (and therefore its chances of firing) critically depends on the efficacy of the synapses. Each synapse actually contains a gap, with neurotransmitter chemicals poised to transmit a signal across the gap. One of the most influential researchers into neurological systems (Donald Hebb) postulated that learning consisted principally in altering the "strength" of synaptic connections. For example, in the classic Pavlovian conditioning experiment, where a bell is rung just before dinner is delivered to a dog, the dog rapidly learns to associate the ringing of a bell with the eating of food. The synaptic connections between the appropriate part of the auditory cortex and the salivation glands are strengthened, so that when the auditory cortex is stimulated by the sound of the bell the dog starts to salivate. Recent research in cognitive science, in particular in the area of non-conscious information processing, have further demonstrated the enormous capacity of the human mind to infer (learn) simple input-output co-variations from extremely complex stimuli.

Thus, from a very large number of extremely simple processing units (each performing a weighted sum of its inputs, and then firing a binary signal if the total input exceeds a certain level) the brain manages to perform extremely complex tasks. Of course, there is a great deal of complexity in the brain which has not been discussed here, but it is interesting that artificial neural networks can achieve some remarkable results using a model not much more complex than this.

The Basic Artificial Model

To capture the essence of biological neural systems, an artificial neuron is defined as follows:

- It receives a number of inputs (either from original data, or from the output of other neurons in the neural network). Each input comes via a connection that has a strength (or *weight*); these weights correspond to synaptic efficacy in a biological neuron. Each neuron also has a single threshold value. The weighted sum of the inputs is formed, and the threshold subtracted, to compose the activation of the neuron (also known as the post-synaptic potential, or PSP, of the neuron).
- The activation signal is passed through an activation function (also known as a transfer function) to produce the output of the neuron.

If the step activation function is used (i.e., the neuron's output is 0 if the input is less than zero, and 1 if the input is greater than or equal to 0) then the neuron acts just like the biological neuron described earlier (subtracting the threshold from the weighted sum and comparing with zero is equivalent to comparing the weighted sum to the threshold). Actually, the step function is rarely used in artificial neural networks, as will be discussed. Note also that weights can be negative, which implies that the synapse has an inhibitory rather than excitatory effect on the neuron: inhibitory neurons are found in the brain.

This describes an individual neuron. The next question is: how should neurons be connected together? If a network is to be of any use, there must be inputs (which carry the values of variables of interest in the outside world) and outputs (which form predictions, or control signals). Inputs and outputs correspond to sensory and motor nerves such as those coming from the eyes and leading to the hands. However, there also can be hidden neurons that play an internal role in the network. The input, hidden and output neurons need to be connected together.

The key issue here is feedback. A simple network has a feedforward structure: signals flow from inputs, forwards through any hidden units, eventually reaching the output units. Such a structure has stable behavior. However, if the network is recurrent (contains connections back from later to earlier neurons) it can be unstable, and has very complex dynamics. Recurrent networks are very interesting to researchers in neural networks, but so far it is the feedforward structures that have proved most useful in solving real problems.

A typical feedforward network has neurons arranged in a distinct layered topology. The input layer is not really neural at all: these units simply serve to introduce the values of the input variables. The hidden and output layer neurons are each connected to all of the units in the preceding layer. Again, it is possible to define networks that are partially-connected to only some units in the preceding layer; however, for most applications fully-connected networks are better.

When the network is executed (used), the input variable values are placed in the input units, and then the hidden and output layer units are progressively executed. Each of them calculates its activation value by taking the weighted sum of the outputs of the units in the preceding layer, and subtracting the threshold. The activation value is passed through the activation function to produce the output of the neuron. When the entire network has been executed, the outputs of the output layer act as the output of the entire network.

Neural Network Architecture for Solving Quadratic Equations

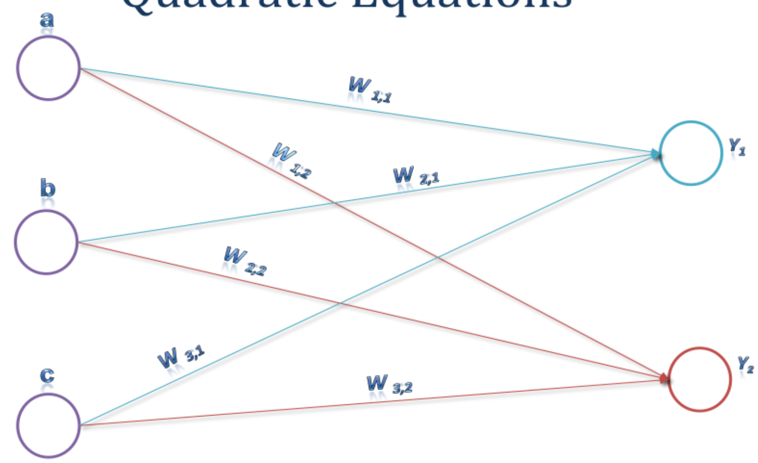


Figure : Solving Quadratic Equation without Hidden NN

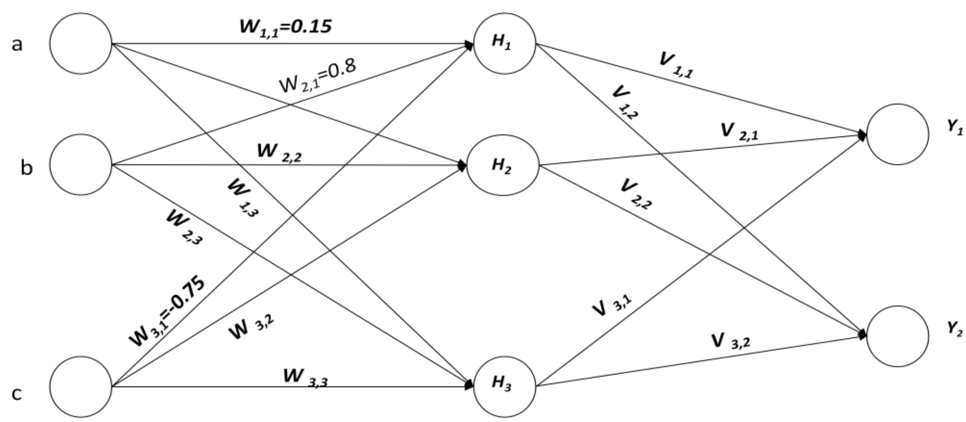


Figure : Solving Quadratic Equation with Hidden NN

<u>To compute the value of the output unit, Y,</u>

1. Place values for a, b and c on the input layer units.

2. Let these values be 1.0, 0.3 and 0.25 as in the above figure.

3. Compute the value of the hidden layer unit, H.

The first step of this computation is to look at each lower level unit that is connected to the hidden unit. For each of these connections, find the value of the unit and multiply by the weight and sum all the results. The calculations give below.

4. Leave the activation value.

Here the linear activation function is used. Back propagation works best when non linear activation functions are used. The most commonly used non-linear function is:

where s = sum of the inputs to the neuron and v is the value of the neuron.

<u>To compute the value of the output unit, Y,</u>

1. Place values for a, b and c on the input layer units.

2. Let these values be 1.0, 0.3 and 0.25 as in the above figure.

3. Compute the value of the hidden layer unit, H.

The first step of this computation is to look at each lower level unit that is connected to the hidden unit. For each of these connections, find the value of the unit and multiply by the weight and sum all the results. The calculations give below.

4. Leave the activation value.

Here the linear activation function is used. Back propagation works best when non linear activation functions are used. The most commonly used non-linear function is:

where s = sum of the inputs to the neuron and v is the value of the neuron.

[Note: This particular function will be called the “standard sigmoid” in this manual. Quite often it is called the “Logistic function”. The general function used to compute the value of a neuron can be called the *activation function, squashing function* or *transfer function.*]

5. The formulas for computing the activation value for a neuron, j can be written more concisely as follows. Let the activation value for neuron j be ‘O_j’. Let the activation function be the general function, f. Let the weight between neuron ‘j’ and neuron ‘i' be ‘w_{ij}’. Let the net input to neuron j be net_j, then

$$net_j = \sum_{i=1,n} W_{ij} O_i$$

where n is the number of units feeding into unit j and

$$O_j = f(net_j)$$

Net input of the first Neuron

$$a \times w_{1,1} + b \times w_{2,1} + c \times w_{3,1} = (1 \times 0.15) + (0.3 \times 0.8) + \big(0.25 \times (-0.75)\big) = 0.2025$$

Net output of the first Neuron (Input of the Hidden Neuron)

$$H_1 = Logsig(0.2025) = \frac{1}{1 + e^{-0.2025}} = 0.5505$$

Net input of the Second Neuron

$$a \times w_{1,2} + b \times w_{2,2} + c \times w_{3,2} = (1 \times (-0.38)) + (0.3 \times 0.43) + \big(0.25 \times (0.7)\big) = -0.076$$

Net output of the Second Neuron (Input of the Hidden Neuron)

$$H_2 = Logsig(-0.076) = \frac{1}{1 + e^{0.076}} = 0.4810$$

Net input of the Third Neuron

$$a \times w_{1,3} + b \times w_{2,3} + c \times w_{3,3} = (1 \times (0.22)) + (0.3 \times (-0.99)) + \big(0.25 \times (0.68)\big) = 0.093$$

Net output of the Third Neuron (Input of the Hidden Neuron)

$$H_3 = Logsig(0.093) = \frac{1}{1 + e^{-0.093}} = 0.5232$$

Output of the First Hidden Neuron

$$H_1 \times V_{1,1} + H_2 \times V_{2,1} + H_3 \times V_{3,1} = (0.5505 \times 0.6) + (0.4810 \times 0.79) + (0.5232 \times (-0.35))$$
$$= 0.5272$$

Output of the Second Hidden Neuron

$$H_1 \times V_{1,2} + H_2 \times V_{2,2} + H_3 \times V_{3,2} = (0.5505 \times 0.8) + (0.4810 \times 0.49) + (0.5232 \times (-0.65))$$
$$= 0.3360$$

Error= Target – Actual Output

Training:

The training process will try to adjust the weights so that the answers come out right.

1. Put one of the patterns to be learned on the input units.

2. Find the values for the hidden unit and output unit.

3. Find out how large the error is on the output unit.

4. Use one of the back-propagation formulas to adjust the weights leading into the output unit. The idea is to try to make the answer come out just a little bit closer to the right answer.

5. Use another formula to find out errors for the hidden layer unit.

6. Adjust the weights leading into the hidden layer unit via another formula.

7. Repeat steps one thru six for Quadratic Equation Co- efficient patterns.

8. Even after all these changes to the weights the answers will only be a little closer to the right answers and the whole process has to be repeated many times. Each time all the patterns in the problem have been used once we will call that an iteration although often other people call this an “epoch”.

Neuron Model and Network Architectures

Objectives:

In this section we presented a simplified description of biological neurons and neural networks. Now we will introduce our simplified mathematical model of the neuron and will explain how these artificial neurons can be interconnected to form a variety of network architectures. We will also illustrate the basic operation of these networks through some simple examples. The concepts and notation introduced in this chapter will be used throughout this book.

This section does not cover all of the architectures that will be used in this book, but it does present the basic building blocks. More complex architectures will be introduced and discussed as they are needed in later chapters. Even so, a lot of detail is presented here. Please note that it is not necessary for the reader to memorize all of the material in this chapter on a first reading. Instead, treat it as a sample to get you started and a resource to which you can return.

Notation:

Neural networks are so new that standard mathematical notation and architectural representations for them have not yet been firmly established. In addition, papers and books on neural networks have come from many diverse fields, including engineering; physics, psychology and mathematics, and many authors tend to use vocabulary peculiar to their specialty. As a result, many books and papers in this field are difficult to read, and concepts are made to seem more complex than they actually are. This is a shame, as it has prevented the spread of important new ideas. It has also led to more than one "reinvention of the wheel."

In this book we have tried to use standard notation where possible, to be clear and to keep matters simple without sacrificing rigor. In particular, we have tried to define practical conventions and use them consistently.

Figures, mathematical equations and text discussing both figures and mathematical equations will use the following notation:

Scalars - small *italic* letters: ***a,b,c***

Vectors - small **bold** nonitalic letters: **a,b,c**

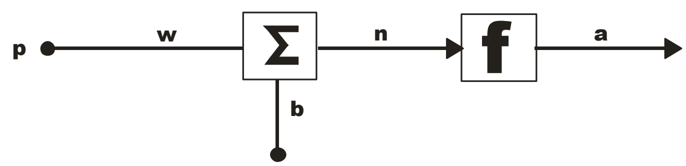
Matrices - capital **BOLD** nonitalic letters: **A,B,C**

Additional notation concerning the network architectures will be introduced as you read this chapter. A complete list of the notation that we use throughout the book is given in: Appendix B, so you can look there if you have a question.

Neuron Model

Single-Input Neuron:

A single-input neuron is shown the figure given below. The scalar input *p* is multiplied by the scalar *weight* w to form *wp,* one of the terms that is sent to the summer. The other input, 1, is multiplied by *a bias b* and then passed, to the summer. The summer output n, often referred to as the *net input,* goes into a *transfer function f,* which produces the scalar neuron output *a.* (Some authors use the term "activation function" rather than *transfer function* and "offset" rather than *bias.)*
If we relate this simple model back to the biological neuron that we discussed, the weight w corresponds to the strength of a synapse, the cell body is represented by the summation and the transfer function, and the neuron output *a* represents the signal on the axon.



a=f(wp+b)

The neuron output is calculated as

$$a = f(wp + b)$$

If, for instance, *w* = 3, *p* = 2 and *b* = -1.5, then

$$a = f(3(2) - 1.5) = f(4.5)$$

The actual output depends on the particular transfer function that is chosen. We will discuss transfer functions in the next section.

The bias is much like a weight, except that it has a constant input of 1. However, if you do not want to have a bias in a particular neuron, it can be omitted. We will see examples this in chapters An illustrative example, Supervised Hebbian Learning and Competitive network.

Note that w and b are both *adjustable* scalar parameters of the neuron. Typically the transfer function is chosen by the designer and then the parameters *w* and *b* will be adjusted by some learning rule so that the neuron input/output relationship meets some specific goal (see Perceptron Learning rule for an introduction to learning rules). As described in the following section, we have different transfer functions for different purposes.

Transfer Functions:

The transfer function in above figure may be a linear or a nonlinear function of n. A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve.

A variety of transfer functions have been included in this book. Three of the most commonly used functions are discussed below.

The *hard limit transfer function,* shown on the left side of the following figure, sets the output of the neuron to 0 if the function argument is less than 0, or 1 if its argument is greater than or equal to O. We will use this function to create neurons that classify inputs into two distinct categories. It will be used extensively in Perceptron Learning rule.

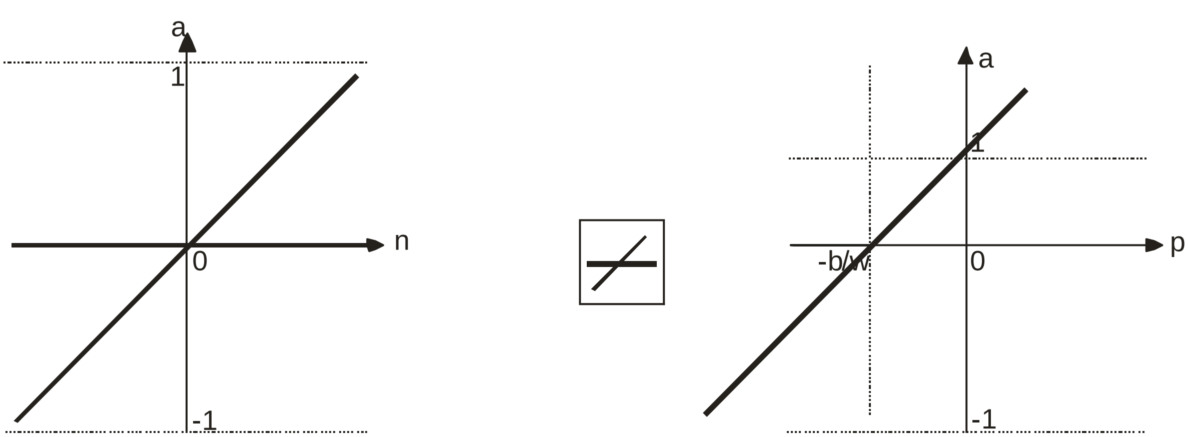
The graph on the right side of the following figure illustrates the input/output characteristic of a single-input neuron that uses a hard limit transfer function. Here we can see the effect of the weight and the bias. Note that an icon for the hard limit transfer function is shown between the two figures. Such icons will replace the general *f* in network diagrams to show the particular transfer function that is being used.

The output of a *linear transfer function* is equal to its input:

$$a = n \tag{1}$$

as illustrated in the following figure..

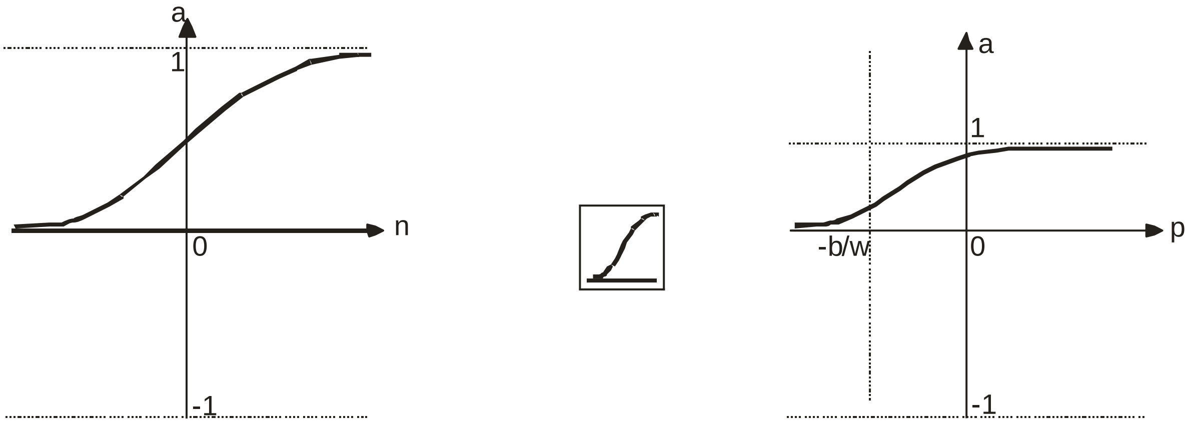
Neurons with this transfer function are used in the ADALlNE networks, which are discussed in Adaptive Learning.



a=purelim(n) a=purelim(wp+b)

The output *(a*) versus input *(p*) characteristic of a single-input linear neuron with a bias is shown on the right of above figure.

The log-sigmoid transfer function is shown in given figure below .



a=logsig(n) a=logsig(wp+b)

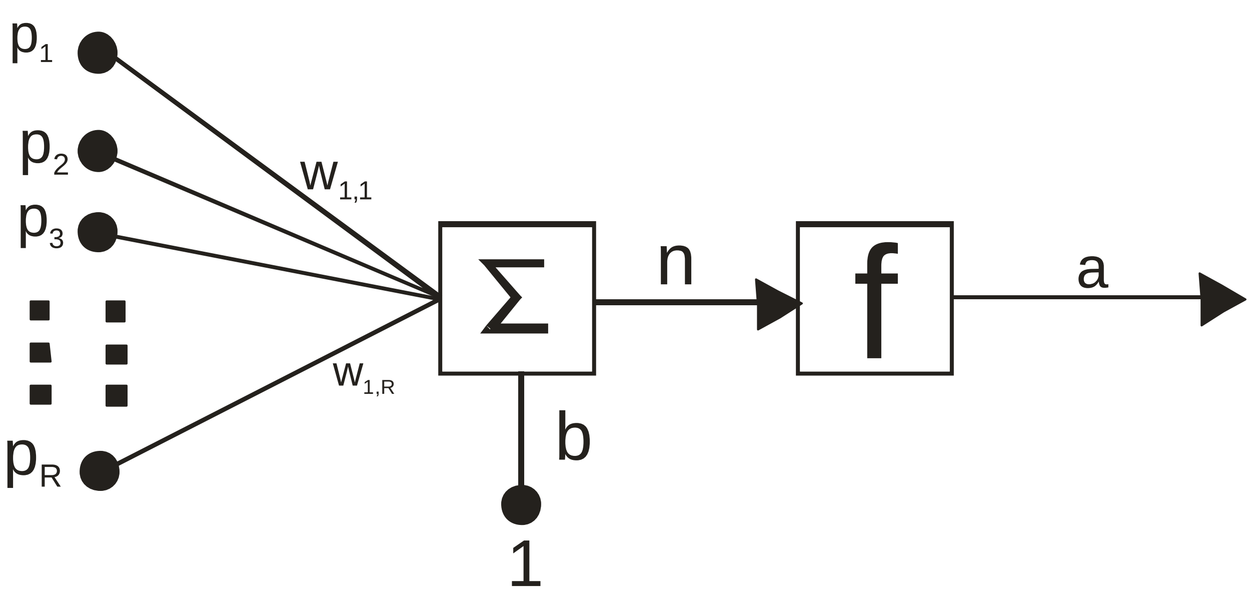
This transfer function takes the input (which may have any value between plus and minus infinity) and squashes the output into the range 0 to 1, according to the expression:

$$a = \frac{1}{1+e^{-n}}. \tag{2}$$

The log-sigmoid transfer function is commonly used in multilayer networks that are trained using the backpropagation algorithm, in part because this function is differentiable (see Using Backpropagation).

Multiple-Input Neuron

Typically, a neuron has more than one input. A neuron with R inputs is shown in the following figure. The individual inputs $p_1, p_2, \cdots, p_R$ are each weighted by corresponding elements $W_{1,1}, W_{1,2}, \cdots W_{1,R}$ of the *weight matrix* W.



a=f(wp+b)

The neuron has a bias *b*, which is summed with the weighted inputs to form the net input n:

$$n = W_{1,1}p_1 + W_{1,2}p_2 + \cdots + W_{1,R}p_R + b \tag{3}$$

This expression can be written in matrix form:

$$n = Wp + b \tag{4}$$

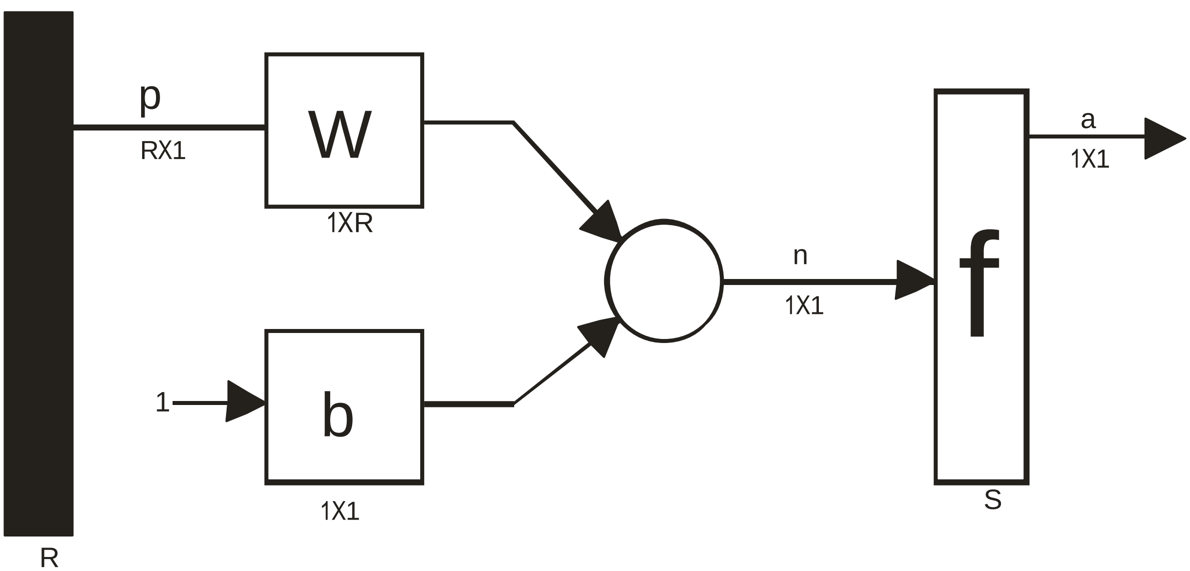
where the matrix W for the single neuron case has only one row. Now the neuron output can be written as

$$a = f(Wp + b) \tag{5}$$

Fortunately, neural networks can often be described with matrices. This kind of matrix expression will be used throughout the book. Don't be concerned if you are rusty with matrix and vector operations. we will provide many examples and solved problems that will spell out the procedures.

We have adopted a particular convention in assigning the indices of the elements of the weight matrix. The first index indicates the particular neuron destination for that weight. The second index indicates the source of the signal fed to the neuron. Thus, the indices in $W_{1,2}$ say that this weight represents the connection *to* the first (and only) neuron *from* the second source. Of course, this convention is more useful if there is more than one neuron, as will be the case later in this section.

We would like to draw networks with several neurons, each having several inputs. Further, we would like to have more than one layer of neurons. You can imagine how complex such a network might appear if all the lines were drawn. It would take a lot of ink, could hardly be read, and the mass of detail might obscure the main features. Thus, we will use an abbreviated *notation.* A multiple-input neuron using this notation is shown in below figure.



As shown in above figure, the input vector p is represented by the solid vertical bar at the left. The dimensions of p are displayed below the variable as$R \times 1$, indicating that the input is a single vector of R

elements. These inputs go to the weight matrix W, which has *R* columns but only one row in this single neuron case. 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*, which is the sum of the bias *b* and the product **Wp.** The neuron's output *a* is a scalar in this case. If we had more than one neuron, the network output would be a vector.

The dimensions of the variables in these abbreviated notation figures will always be included, so that you can tell immediately if we are talking about a scalar, a vector or a matrix. You will not have to guess the kind of variable or its dimensions.

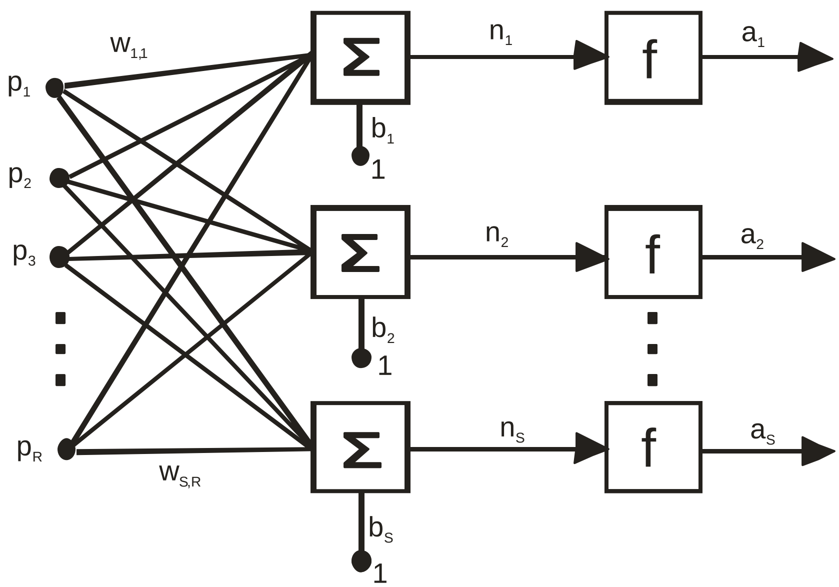
Note that the number of inputs to a network is set by the external specifications of the problem. If, for instance, you want to design a neural network that is to predict kite-flying conditions and the inputs are air temperature, wind velocity and humidity, then there would be three inputs to the net work.

Network Architectures

Commonly one neuron, even with many inputs, may not be sufficient. We might need five or ten, operating in parallel, in what we will call a “layer”. This concept of a layer is discussed below.

A Layer of Neurons

A *single-layer* network of S neurons is shown in given below figure. Note that each of the *R* inputs is connected to each of the neurons and that the weight matrix now has S rows.



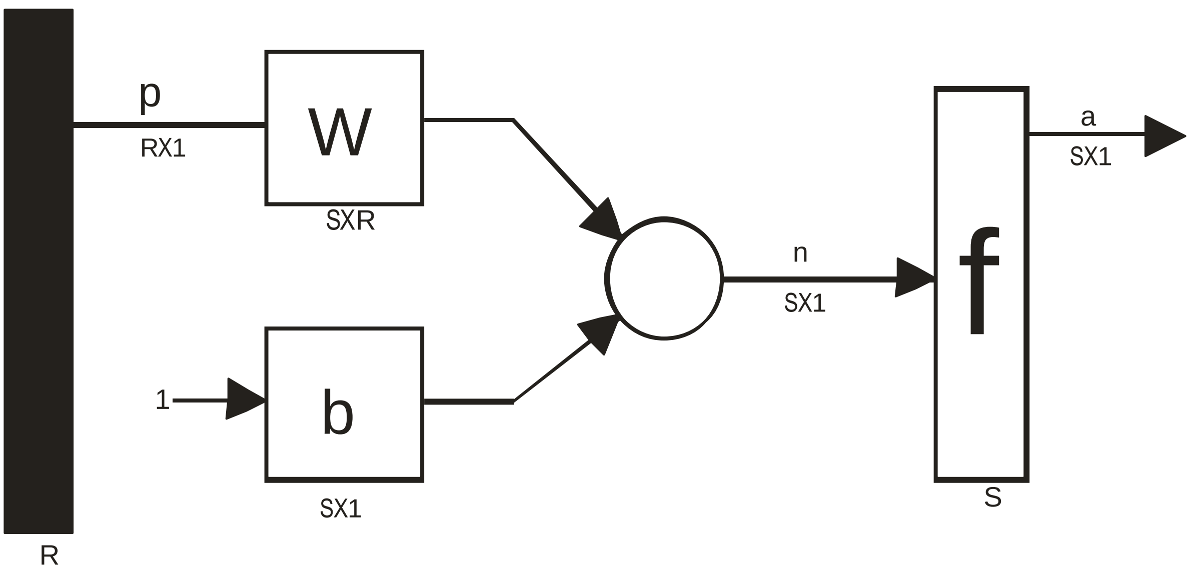
The layer includes the weight matrix, the summers, the bias vector b, the transfer function boxes and the output vector a. Some authors refer to the inputs as another layer, but we will not do that here.

Each element of the input vector p is connected to each neuron through the weight matrix W. Each neuron has a bias b_i, a summer, a transfer function *f* and an output *a*. Taken together, the outputs form the output vector a.

It is common for the number of inputs to a layer to be different from the number of neurons that if $R \neq S$. You might ask if all the neurons in a layer must have the same transfer function. The answer is no; you can define a single (composite) layer of neurons having different transfer functions by combining two of the networks shown above 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:
As noted previously, the row indices of the elements of matrix W indicate the destination neuron associated with that weight, while the column indices indicate the source of the input for that weight. Thus, the indices in $w_{3,2}$ say that this weight represents the connection *to* the third neuron *from* the second source.

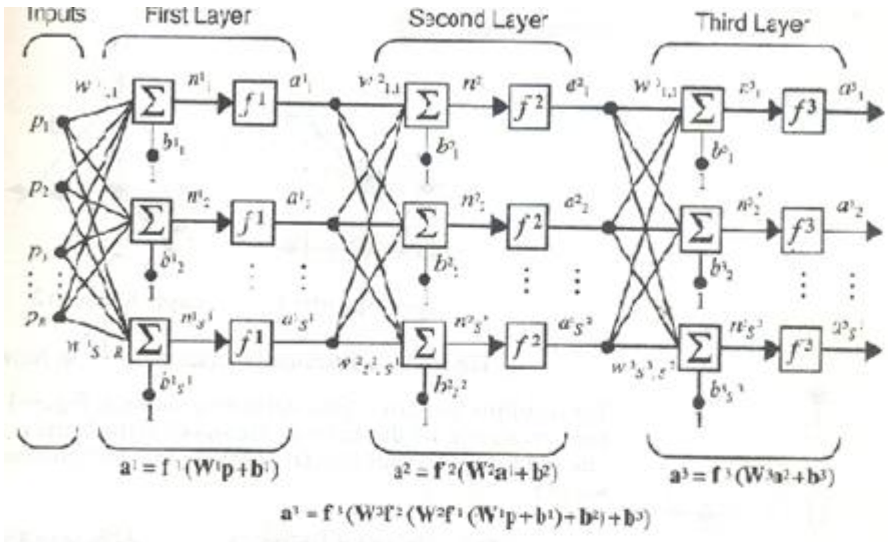
Fortunately, the S-neuron, R-input, one-layer network also can be drawn in abbreviated notation, as shown in the figure given below.



Here again, the symbols below the variables tell you that for this layer, p is a vector of length *R*, W is an S x *R* matrix, and a and b are vectors of length S. As defined previously, the layer includes the weight matrix, the summation and multiplication operations, .the bias vector b, the transfer function boxes and the output vector.

Multiple Layers of Neurons

Now consider a network with several layers. Each layer has its own weight matrix W, its own bias vector b, a net input vector D and an output vector a. We need to introduce some additional notation to distinguish between these layers. We will use superscripts to identify the layers. Specifically, we append the number of the layer as a *superscript* to the names for each of these variables. Thus, the weight matrix for the first layer is written as W^1, and the weight matrix for the second layer is written as W^2. This notation is used in the three-layer network shown in the following figure.



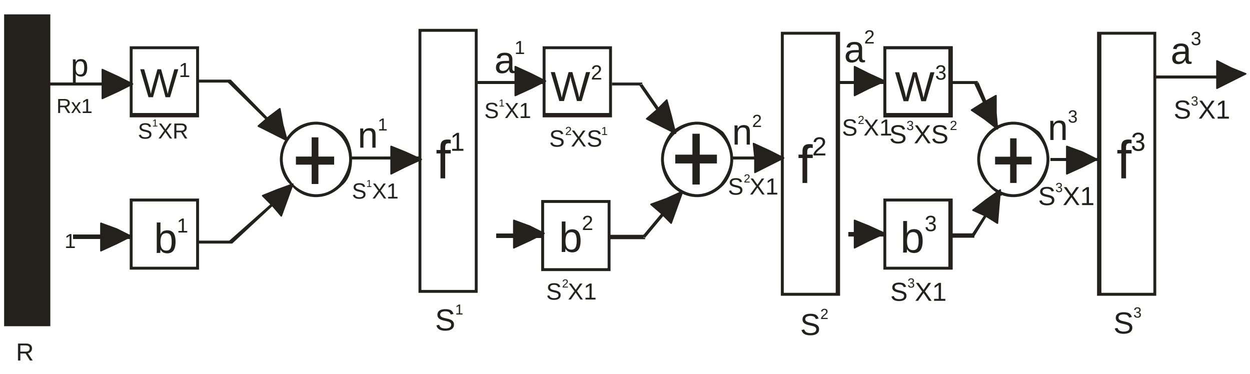
As shown, there are *R* inputs, S^1 neurons in the first layer, S^2 neurons in

the second layer, etc. As noted, different layers can have different numbers of neurons.

The outputs of layers one and two are the inputs for layers two and three. Thus layer 2 can be viewed as a one-layer network with $R = S^1$ inputs, $S = S^2$ neurons, and an $S^1 \times S^2$ weight matrix W^2. The input to layer 2 is a^1, and the output is a^2.

A layer whose output is the network output is called an *output layer.* The other layers are called *hidden layers.* The network shown above has an output layer (layer 3) and two hidden layers (layers 1 and 2).

The same three-layer network discussed previously also can be drawn using our abbreviated notation, as shown in the following figure.



Multilayer networks are more powerful than single-layer networks. For instance, a two-layer network having a sigmoid first layer and a linear second layer can be trained to approximate most functions arbitrarily well. Single-layer networks cannot do this.

At this point the number of choices to be made in specifying a network may look overwhelming, so let us consider this topic. The problem is not as bad as it looks. First, recall that the number of inputs to the network and the number of outputs from the network are defined by external problem specifications. So if there are four external variables to be used as inputs, there are four inputs to the network. Similarly, if there are to be seven outputs from the network, there must be seven neurons in the output layer. Finally, the desired characteristics of the output signal also help to select the transfer function for the output layer. If an output is to be either –l or 1, then a symmetrical hard limit transfer function should be used. Thus, the architecture of a single-layer network is almost completely determined by problem specifications, including the specific number of inputs and outputs and the particular output signal characteristic.

Now, what if we have more than two layers? Here the external problem does not tell you directly the number of neurons required in the hidden layers. In fact, there are few problems for which one can predict the optimal number of neurons needed in a hidden layer. This problem is an active area of research. We will develop some feeling on this matter as we proceed to using Backpropagation, Backpropagation.

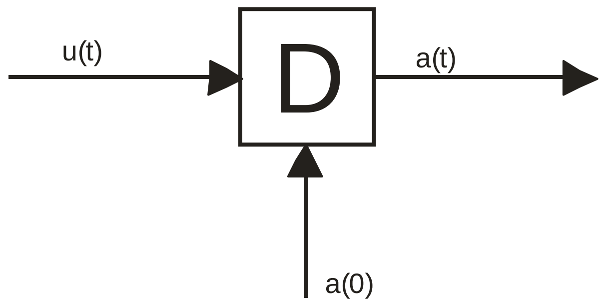
As for the number of layers, most practical neural networks have just two or three layers. Four or more layers are used rarely.

We should say something about the use of biases. One can choose neurons with or without biases. The bias gives the network an extra variable, and so you might expect that networks with biases would be more powerful than those without, and that is true. Note, for instance, that a neuron without a bias will always have a net input *n* of zero when the network inputs p are zero. This may not be desirable and can be avoided by the use of a bias. The effect of the bias is discussed more fully in An illustrative example and Perceptron Learning Rule.

In later we will omit a bias in some examples or demonstrations. In some cases this is done simply to reduce the number of network parameters. With just two variables, we can plot system convergence in a two-dimensional plane. Three or more variables are difficult to display.

Recurrent Networks

Before we discuss recurrent networks, we need to introduce some simple building blocks. The first is the *delay* block which is illustrated in the following figure.



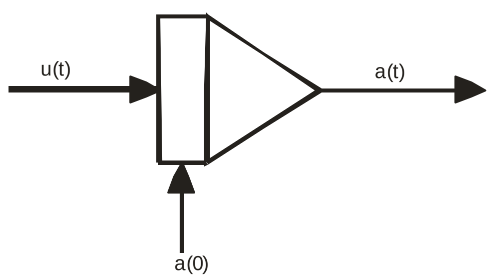
a(t)=u(t-1)

The delay output a(t) is computed from its input u(t) according to

$$a(t) = u(t-1) \tag{7}$$

Thus the output is the input delayed by one time step. (This assumes that time is updated in discrete steps and takes on only integer values.) Eq. (7) requires that the output be initialized at time $t = 0$. This initial condition is indicated in above figure by the arrow coming into the bottom of the delay block.

Another related building block, which we will use for the continuous-time recurrent networks in Grossberg Network, Adaptive Resonance theory, Stability and Hopfield Network, is the *integrator*, which is shown in figure given below.

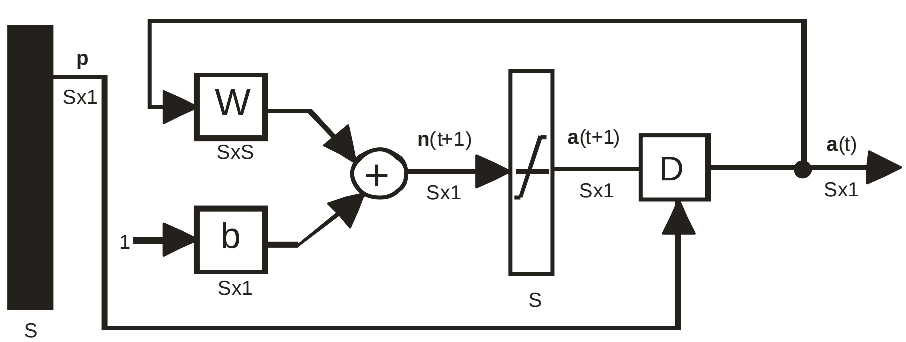


The integrator output a(t) is computed from its input u(t) according to

$$a(t) = \int_0^t u(\tau)d\tau + a(0). \tag{8}$$

The initial condition a(0) is indicated by the arrow coming into the bottom of the integrator block.

We are now ready to introduce recurrent networks. A *recurrent network* is a network with feedback; some of its outputs are connected to its inputs. This is quite different from the networks that we have studied thus far, which were strictly feedforward with no backward connections. One type of discrete-time recurrent network is shown in the following figure:



In this particular network the vector p supplies the initial conditions ($i.e.\ a\,(0)\ =\ p$). Then future outputs of the network are computed from previous outputs:

$$a(1) = satlins(Wa(0) + b), \qquad a(2) = satlins(Wa(1) + b), \cdots$$

Recurrent networks are potentially more powerful than feedforward net works and can exhibit temporal behavior. These types of networks are discussed in An illustrative example and Grossberg Network, Adaptive Resonance theory, Stability and Hopfield Network.

Problem Statement:

A produce dealer has a warehouse that stores a variety of fruits and vegetables. When fruit is brought to the warehouse, various types of fruit may be mixed together. The dealer wants a machine that will sort the fruit according to type. There is a conveyer belt on which the fruit is loaded. This conveyer passes through a set of sensors, which measure three properties of the fruit: *shape*, *texture* and *weight*. These sensors are somewhat primitive. The shape sensor will output a 1 if the fruit is approximately round and a-1 if it is more elliptical. The texture sensor will output a 1 if the surface of the fruit is smooth and a -1 if it is rough. The weight sensor will output a 1 if the fruit is more than one pound and a-1 if it is less than one pound.

The three sensor outputs will then be input to a neural network. The purpose of the network is to decide which kind of fruit is on the conveyor, so that the fruit can be directed to the correct storage bin. To make the problem even simpler, let's assume that there are only two kinds of fruit on the conveyor: apples and oranges.

As each fruit passes through the sensors it can be represented by a three dimensional vector. The first element of the vector will represent shape, the second element will represent texture and the third element will represent weight:

$$p = \begin{bmatrix} shape \\ texture \\ weight \end{bmatrix}. \tag{9}$$

Therefore, a prototype orange would be represented by

$$p_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \tag{10}$$

and a prototype apple would be represented by

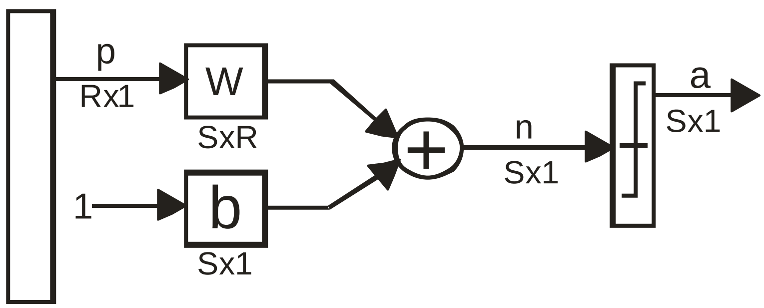
$$p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}. \tag{11}$$

The neural network will receive one three-dimensional input vector for each fruit on the conveyer and must take a decision as to whether the fruit is an *orange* (p_1) or an *apple*. (p_2).

Now that we have defined this simple (trivial?) pattern recognition problem, let's look briefly at three different neural networks that could be used to solve it. The simplicity of our problem will facilitate our understanding of the operation of the networks.

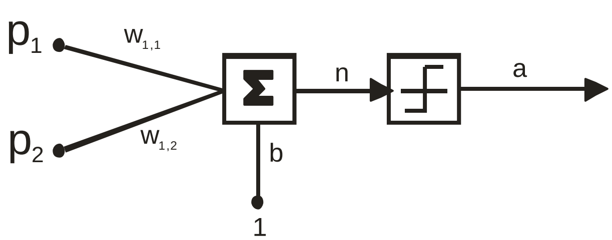
Perceptron

The first network we will discuss is the perceptron. The following figure illustrates a single-layer perceptron with a symmetric hard limit transfer function *hard-lims.*



Two-Input Case

Before we use the perceptron to solve the orange and apple recognition problem (which will require a three-input perceptron, i.e., *R* = 3), it is useful to investigate the capabilities of a two-input/single-neuron perceptron *(R* = 2), which can be easily analyzed graphically. The two-input perceptron is shown in figure given below.



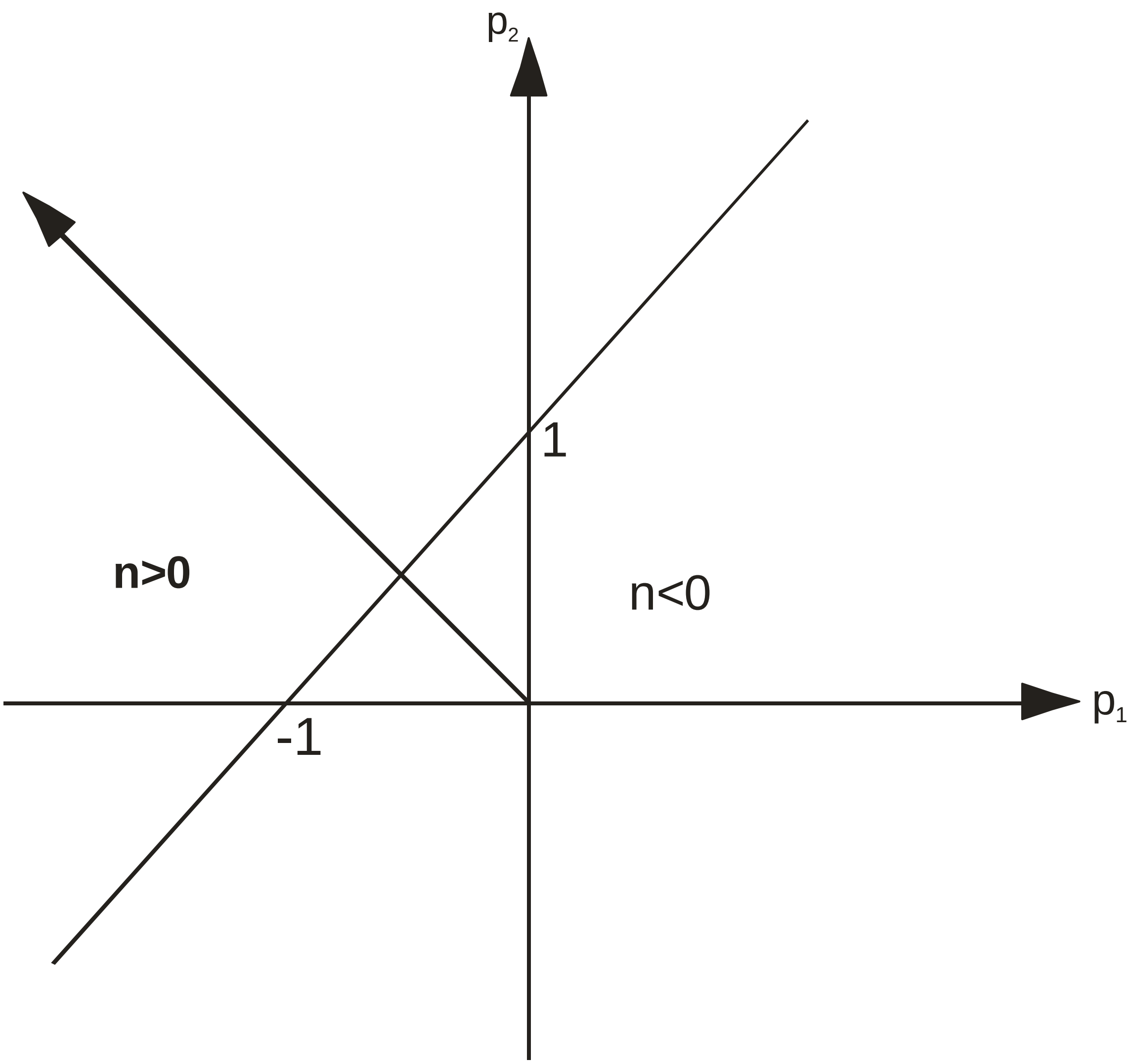
Single-neuron perceptrons can classify input vectors into two categories. For example, for a two-input perceptron, if $w_{1,1} = -1$ and $w_{1,2} = 1$ then

$$a = hardlims\ (n) = hardlims\ ([-1\ 1]\ P + b). \qquad (12)$$

Therefore, if the inner product of the weight matrix (a single row vector in this case) with the input vector is greater than or equal to -*b*, the output will be 1. If the inner product of the weight vector and the input is less than -*b*, the output will be -1. This divides the input space into two parts. The given below figure illustrates this for the case where *b* = -1. The blue line in the figure represents all points for which the net input *n* is equal to 0:

$$n = [-1\ 1]p - 1 = 0. \qquad (13)$$

Notice that this decision boundary will always be orthogonal to the weight matrix, and the position of the boundary can be shifted by changing *b*. (In the general case, W is a matrix consisting of a number of row vectors, each of which will be used in an equation like Eq. (13). There will be one boundary for each row of W. See Perceptron Architecture for more on this topic.) The shaded region contains all input vectors for which the output of the network will be 1. The output will be -1 for all other input vectors.



The key property of the single-neuron perceptron, therefore, is that it can separate input vectors into two categories. The decision boundary between the categories is determined by the equation

$$Wp + b = 0. \tag{14}$$

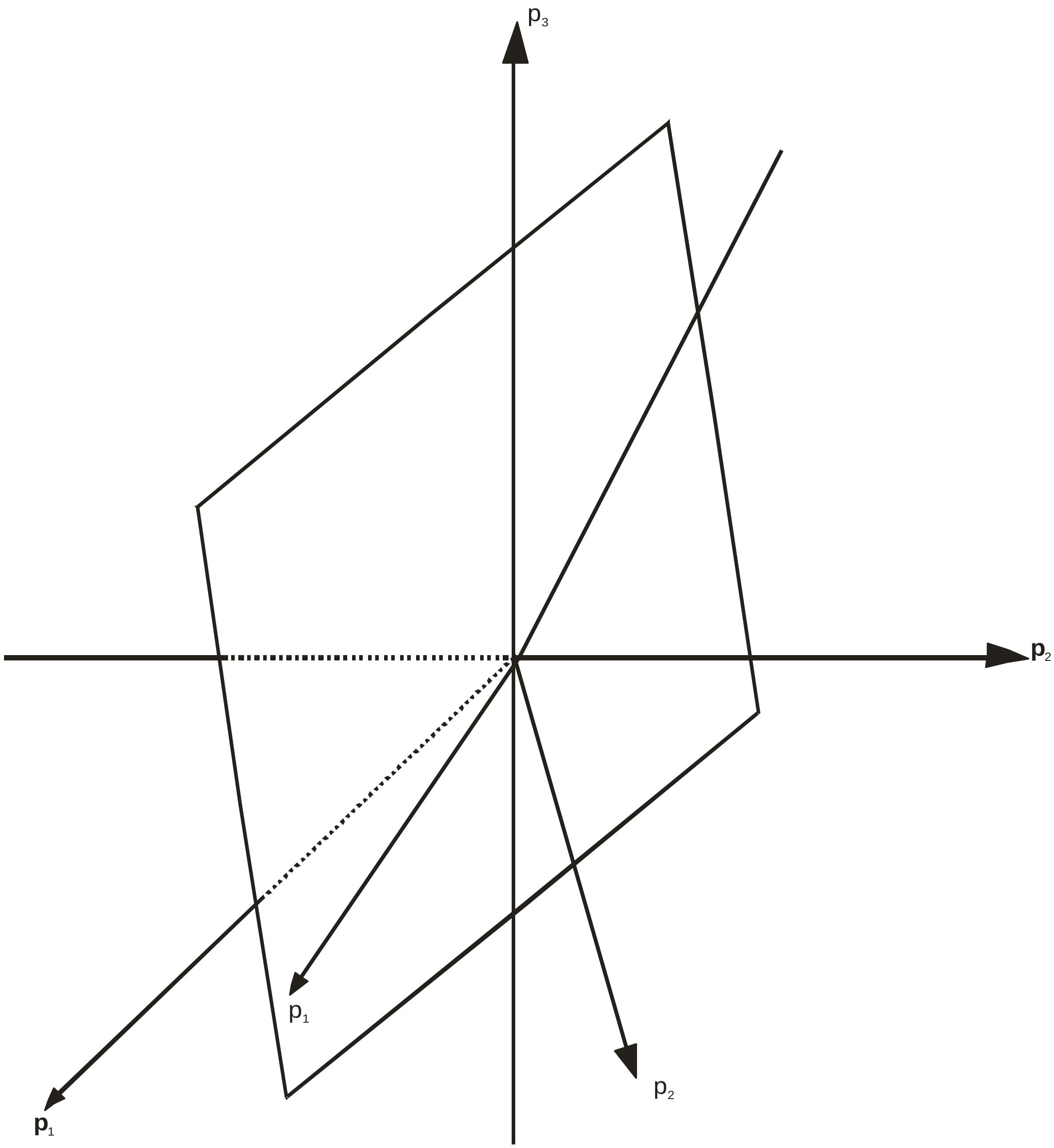
Because the boundary must be linear, the single-layer perceptron can only be used to recognize patterns that are linearly separable (can be separated by a linear boundary). These concepts will be discussed in more detail in Perceptron Architecture.

Pattern Recognition Example

Now consider the apple and orange pattern recognition problem. Because there are only two categories, we can use a single-neuron perceptron. The vector inputs are three-dimensional *(R* = 3), therefore the perceptron equation will be

$$a = \text{hardlims}\left[\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + b\right] \tag{15}$$

We want to choose the bias *b* and the elements of the weight matrix so that the perceptron will be able to distinguish between apples and oranges. For example, we may want the output of the perceptron to be 1 when an apple is input and -1 when an orange is input. Using the concept illustrated in above figure, let's find a linear boundary that can separate oranges and apples. The two prototype vectors (recall Eq. (10) and Eq. (11) are shown in given below figure. From this figure we can see that the linear boundary that divides these two vectors symmetrically is the p_1,p_3 plane.



The p_1,p_3 plane, which will be our decision boundary, can be described by the equation

$$p_2 = 0, \tag{16}$$

or

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + 0 = 0. \tag{17}$$

Therefore the weight matrix and bias will be

$$w = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, \; b = 0. \tag{18}$$

The weight matrix is orthogonal to the decision boundary and points toward the region that contains the prototype pattern p_2 (apple) for which we want to perceptron to produce an output of 1. The bias is 0 because the decision boundary passes through the origin.

Now let’s test the operation of our perceptron pattern classifier. It classifies perfect apples and oranges correctly since

Orange:

$$a = \text{hardlims}\left[\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0 \right] = -1(\text{orange}) \tag{19}$$

Apple:

$$a = \text{hardlims}\left[\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0 \right] = 1(\text{apple}). \tag{20}$$

But what happens if we put a not-so-perfect orange into the classifier? Let's say that an orange with an elliptical shape is passed through the sensors. The input vector would then be

$$\mathrm{p} = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}. \tag{21}$$

The response of the network would be

$$a = \text{hardlims}\left[\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + 0 \right] = -1(\text{orange}). \tag{22}$$

In fact, any input vector that is closer to the orange prototype vector than to the apple prototype vector (in Euclidean distance) will be classified as an orange (and vice versa).

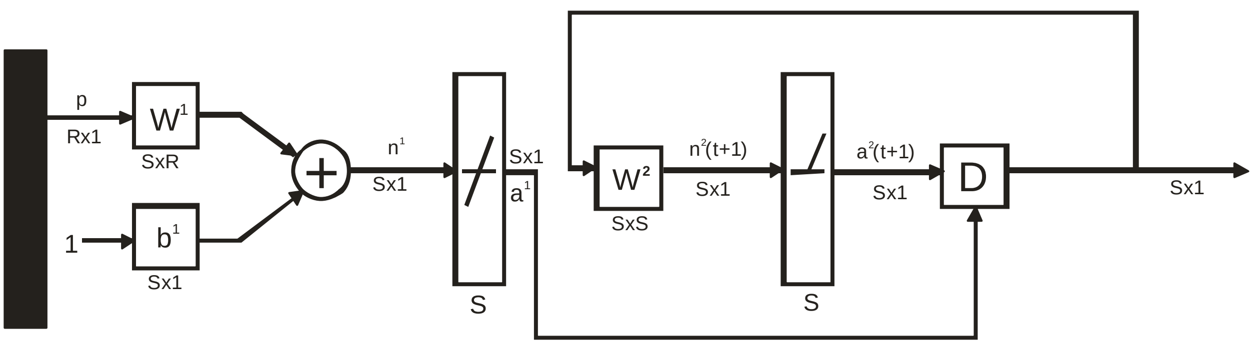
This example has demonstrated some of the features of the perceptron network, but by no means have we exhausted our investigation of perceptrons. This network and variations on it. In the apple/orange example we were able to design a network graphically, by choosing a decision boundary that clearly separated the patterns. What about practical problems, with high dimensional input spaces? The key characteristic of the single-layer perceptron is that it creates linear decision boundaries to separate categories of input vector. What if we have categories that cannot be separated by linear boundaries?

This question will be addressed in Backpropagation, where we will introduce the multilayer perceptron. The multilayer networks are able to solve classification problems of arbitrary complexity.

Hamming Network

The next network we will consider is the Hamming network. It was designed explicitly to solve binary pattern recognition problems (where each element of the input vector has only two possible values - in our example 1 or -1). This is an interesting network, because it uses both feedforward and recurrent (feedback) layers, which were both described in Neuron Model and Network Architectures in given the following figure shows the standard Hamming network. Note that the number of neurons in the first layer is the same as the number of neurons in the second layer.

The objective of the Hamming network is to decide which prototype vector is closest to the input vector. This decision is indicated by the output of the recurrent layer. There is one neuron in the recurrent layer for each prototype pattern. When the recurrent layer converges, there will be only one neuron with nonzero output. This neuron indicates the prototype pattern that is closest to the input vector. Now let's investigate the two layers of the Hamming network in detail.



Feedforward Layer

The feedforward layer performs a correlation, or inner product, between each of the prototype patterns and the input pattern (as we will see in Eq. (25)). In order for the feedforward layer to perform this correlation, the rows of the weight matrix in the feedforward layer, represented by the connection matrix w^1, are set to the prototype patterns. For our apple and orange example this would mean

$$w^1 = \begin{bmatrix} p_1^T \\ p_2^T \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix}. \qquad (23)$$

The feedforward layer uses a linear transfer function, and each element of the bias vector is equal to R , where R is the number of elements in the input vector. For our example the bias vector would be

$$b^1 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}. \qquad (24)$$

With these choices fur the weight matrix and bias vector, the output of the feedforward layer is

$$a^1 = w^1 p + b^1 = \begin{bmatrix} p_1^T \\ p_2^T \end{bmatrix} p + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} p_1^T p + 3 \\ p_2^T p + 3 \end{bmatrix}. \quad (25)$$

Note that the outputs of the feedforward layer are equal to the inner products of each prototype pattern with the input, plus R. For two vectors of equal length (norm), their inner product will be largest when the vectors point in the same direction, and will the smallest when they point in opposite direction. By adding R to the inner product we guarantee that the outputs of the feedforward layer can never be negative. This is required for proper operation of the recurrent layer.

This network is called the Hamming network because the neuron in the feedforward layer with the largest output will correspond to the prototype pattern that is closest in Hamming distance to the input pattern. (The Hamming distance between two vectors is equal to the number of elements that are different. It is defined only for binary vectors.) We leave it to the reader to show that the outputs of the feedforward layer are equal to *2R* minus twice the Hamming distances from the prototype patterns to the input pattern.

Recurrent Layer

The recurrent layer of the Hamming network is what is known as a "competitive" layer. The neurons in this layer are initialized with the outputs of the feedforward layer, which indicate the correlation between the prototype patterns and the input vector. Then the neurons compete with each other to determine a winner. After the competition, only one neuron will have a nonzero output. The winning neuron indicates which category of input was presented to the network (for our example the two categories are *apples* **and *oranges*). The** equations that describe the competition are:

$$a^2(0) = a^1 \ (initial\ condition), \quad (26)$$

and

$$a^2(t+1) = poslin\left(w^2 a^2(t)\right). \quad (27)$$

(Don't forget that the superscripts here indicate the layer number, not a power of 2.) The *poslin* transfer function is linear for positive values and zero for negative values. The weight matrix w^2 has the form

$$w^2 = \begin{bmatrix} 1 & \varepsilon \\ -\varepsilon & 1 \end{bmatrix}, \quad (28)$$

where ε is some number less than 1/ (S - 1) , and S is the number of neurons in the recurrent layer. (Can you show why ε must be less than 1/(S-1)?)

An iteration of the recurrent layer proceeds as follows:

$$a^2(t+1) = poslin\left(\begin{bmatrix} 1 & -\varepsilon \\ -\varepsilon & 1 \end{bmatrix} a^2(t)\right) = poslin\left(\begin{bmatrix} a^2(t) & -\varepsilon a^2(t) \\ a^2(t) & -\varepsilon a^2(t) \end{bmatrix}\right) \quad (29)$$

Each element id reduced by the same fraction of the other. The larger element will be reduced by less, and the smaller element will be reduced by more, therefore the difference between large and small will be increased. The effect of the recurrent layer is to zero out all neuron outputs, except the one with the largest initial value (which corresponds to the prototype pattern that is closest in Hamming distance to the input).

To illustrate the operation of the Hamming network, consider again the oblong orange that we used to test the perceptron:

$$\mathrm{p} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}. \tag{30}$$

The output of the feedforward layer will be

$$a^1 = \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 1+3 \\ -1+3 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \tag{31}$$

The weight matrix for the recurrent layer will be given by Eq. (3.20) with $\varepsilon = 1/2$ (any number less than 1 would work). The first iteration of the recurrent layer produces

$$a^2(1) = poslin\left(W^2 a^2(0)\right) = \begin{cases} poslin\left(\begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix}\right) \\ poslin\left(\begin{bmatrix} 3 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \end{cases}. \tag{32}$$

The second iteration produces

$$a^2(2) = poslin\left(W^2 a^2(1)\right) = \begin{cases} poslin\left(\begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix}\right) \\ poslin\left(\begin{bmatrix} 3 \\ -1.5 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 0 \end{bmatrix} \end{cases}. \tag{33}$$

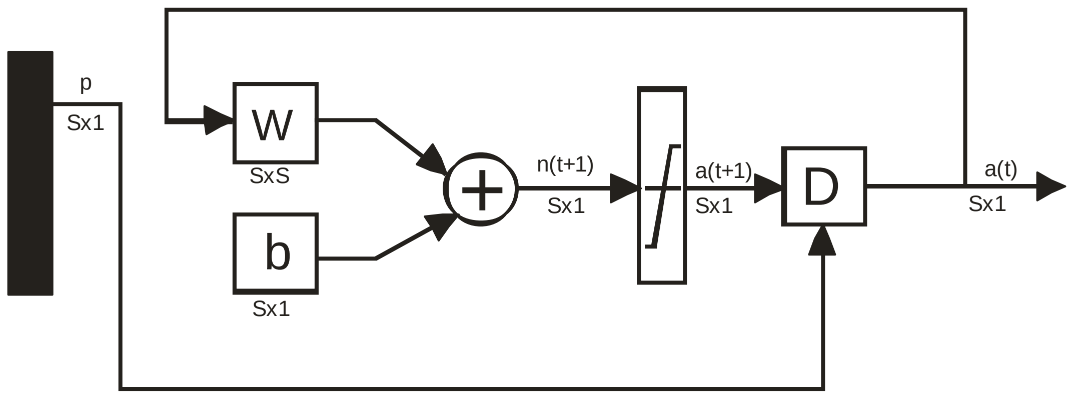
Since the outputs of successive iterations produce the same result, the network has converged. Prototype pattern number one, the *orange*, is chosen as the correct match, since neuron number one has the only nonzero output. (Recall that the first element of a^1 was $(p_1^T + 3)$.) This is the correct choice, since the Hamming distance from the *orange* prototype to the input pattern is 1, and the Hamming distance from the *apple* prototype to the input pattern is 2.

There are a number of networks whose operation is based on the same principles as the Hamming network; that is, where an inner product operation (feedforward layer) is followed by a competitive dynamic layer. They are *self-organizing* networks, which can learn to adjust their prototype vectors based on the inputs that have been presented.

Hopfield Network

The final network we will discuss in this brief preview is the Hopfield network. This is a recurrent network that is similar in some respects to the recurrent layer of the Hamming network, but which can effectively perform the operations of both layers of the Hamming network. A diagram of the Hopfield network is shown in following figure. (This figure is actually a slight variation of the standard Hopfield network. We use this variation because it is somewhat simpler to describe and yet demonstrates the basic concepts.)

The neurons in this network are initialized with the input vector, then the network iterates until the output converges. When the network is operating correctly, the resulting output should be one of the prototype vectors. Therefore, whereas in the Hamming network the nonzero neuron indicates which prototype pattern is chosen, the Hopfield network actually produces the selected prototype pattern at its output.



The equations that describe the network operation are

$$a(0) = p \tag{34}$$

and

$$a(t+1) = satlins(wa(t) + b), \tag{35}$$

where *satlins* is the transfer function that is linear in the range [-1, 1] and saturates at 1 for inputs greater than 1 and at -1 for inputs less than -1.

The design of the weight matrix and the bias vector for the Hopfield network is a more complex procedure than it is for the Hamming network, where the weights in the feedforward layer are the prototype patterns. Hopfield design procedures will be discussed in detail in Hopfield network.

To illustrate the operation of the network, we have determined a weight matrix and a bias vector that can solve our orange and apple pattern recognition problem. They are given in Eq. (36).

$$w = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 1.2 & 0 \\ 0 & 0 & 0.2 \end{bmatrix}, b = \begin{bmatrix} 0.9 \\ 0 \\ -0.9 \end{bmatrix} \tag{36}$$

Although the procedure for computing the weights and biases for the Hopfield network is beyond the scope of this chapter, we can say a few things about why the parameter$ in Eq.(36) work for the apple and orange example.

We want the network output to converge to either the orange pattern, p_1 or the apple pattern, p_2. In both patterns, the first element is 1, and the third element is -1. The difference between the patterns occurs in the second element. Therefore, no matter what pattern is input to the network, we want the first element of the output pattern to converge to l , the third element to converge to -1 , and the second element to go to either 1 or -1 , whichever is closer to the second element of the input vector.

The equations of operation of the Hopfield network, using the parameters given in Eq. (36), are

$$a_1(t+1) = satlins(0.2a_1(t) + 0.9)$$
$$a_2(t+1) = satlins(1.2a_2(t))$$
$$a_3(t+1) = satlins(0.2a_3(t) - 0.9) \tag{37}$$

Regardless of the initial values, $a_i(0)$, the first element will be increased until it saturates at 1 , and the third element will be decreased until it saturates at -1 . The second element is multiplied by a number larger than l. Therefore, if it is initially negative, it will eventually saturate at -1; if it is initially positive it will saturate at 1.

(It should be noted that this is not the only (W, b) pair that could be used. You might want to try some others. See if you can discover what-makes these work.)

Let's again take our oblong orange to test the Hopfield network. The outputs of the Hopfield network for the first three iterations would be

$$a(0) = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}, a(1) = \begin{bmatrix} 0.7 \\ -1 \\ -1 \end{bmatrix}, a(3) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \tag{38}$$

The network has converged to the *orange* pattern, as did both the Hamming network and the perceptron, although each network operated in a different way. The perceptron had a single output, which could take on values of -1 *(orange)* or 1 *(apple)*. In the Hamming network the single nonzero neuron indicated which prototype pattern had the closest match. If the first neuron was nonzero, that indicated *orange*, and if the second neuron was nonzero, that indicated *apple*. In the Hopfield network the prototype pattern itself appears at the output of the network.

As with the other networks demonstrated in this chapter, do not expect to feel completely comfortable with the Hopfield network at this point. There are a number of questions that we have not discussed. For example, "How do we know that the network will eventually converge?" It is possible for recurrent networks to oscillate or to have chaotic behavior. In addition, we have not discussed general procedures for designing the weight matrix and the bias vector.

Perceptron Learning Rule:

Theory and Examples:

In 1943, Warren McCulloch and Walter Pitts introduced one of the first artificial neurons. 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. When the sum is greater than or equal to the threshold, the output is 1. When the sum is less than the threshold, the output is O. They went on to show that networks of these neurons could, in principle, compute any arithmetic or logical function. Unlike biological networks, the parameters of their networks had to be designed, as no training method was available. However, the perceived connection between biology and digital I computers generated a great deal of interest.

In the late 1950s; Frank Rosenblatt and several other researchers developed a class of neural networks-called perceptrons. The neurons in these networks were similar to those of McCulloch and Pitts. Rosenblatt's key contribution was the introduction of a learning rule for training perceptron networks to solve pattern recognition problems. He proved that his learning rule will always converge to the correct network weights, if weights exist that solve the problem. Learning was simple and automatic. Examples of proper behavior were presented to the network, which learned from its mistakes. The perceptron could even learn when initialized with random values for its weights and biases.

Unfortunately, the perceptron network is inherently limited. These limitations were widely publicized in the book *Perceptrons* by Jarvin Minsky and Seymour Papert. They demonstrated that the perceptron networks were incapable of implementing-certain elementary functions. It was not until the 1980s that these limitations were overcome with improved (multilayer) perceptron networks and associated learning rules.

Today the perceptron is still viewed as an important network. It remains a fast and reliable network for the class of problems that it can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks. Thus, the perceptron network and its associated learning rule, are well worth discussion here.

In the remainder of this chapter we will define what we mean by a learning rule, explain the perceptron network and learning rule, and discuss the limitations of the perceptron network.

Learning Rules

As we begin our discussion of the perceptron learning rule, we want to discuss learning rules in general. By *learning rule* we mean a procedure for modifying the weights and biases of a network. (This procedure may also be referred to as a training algorithm.) The purpose of the learning rule is to train the network to perform some task. There are many types of neural network learning rules. They fall into three broad categories: supervised learning, unsupervised learning and reinforcement (or graded) learning.

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

$$\{p_1, t_1\}, \{p_2, t_2\}, \cdots, \{p_Q, t_Q\}, \tag{39}$$

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 weight 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.

Reinforcement learning is similar to supervised learning, except that, instead of being provided with the correct output for each network input, the algorithm is only given a grade. The grade (or score) is a measure of the network performance over some sequence of inputs. This type of learning is currently much less common than supervised learning. It appears to be most suited to control system applications.

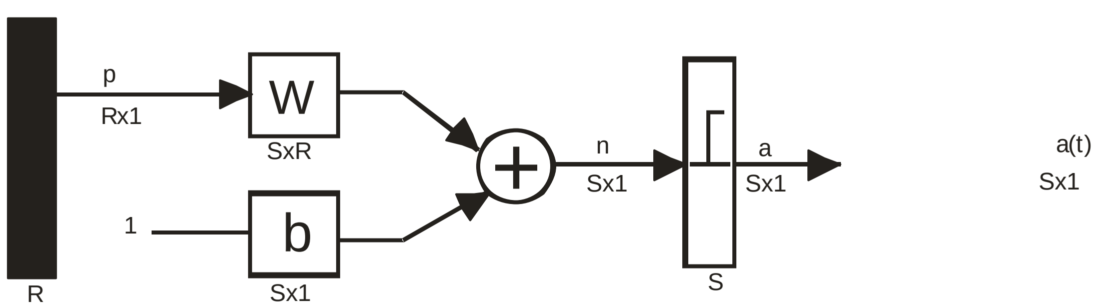
In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. At first glance this might seem to be impractical. How can you train a network if you don't know what it is supposed to do? Most of these algorithms perform some kind of clustering operation. They learn to categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization. There are a number of unsupervised learning algorithms.

Perceptron Architecture

Before we present the perceptron learning rule, let's expand our investigation of the perceptron network, which we began in An illustrative example. The general perceptron network is shown in given below figure.

The output of the network is given by

$$a = hardlim(Wp + b). \tag{40}$$



It will be useful in our development of the perceptron learning rule to be able to conveniently reference individual elements of the network output. Let's see how this can be done. First, consider the network weight matrix:

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & \cdots & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}. \qquad (41)$$

We will define a vector composed of the elements of the ith row of W:

$$w_i = \begin{bmatrix} w_{i,1} \\ w_{i,1} \\ \vdots \\ w_{i,R} \end{bmatrix}. \qquad (42)$$

Now we can partition the weight matrix:

$$W = \begin{bmatrix} W_1^T \\ W_2^T \\ \vdots \\ W_S^T \end{bmatrix}. \qquad (43)$$

This allows us to write the ith element of the network output vector as

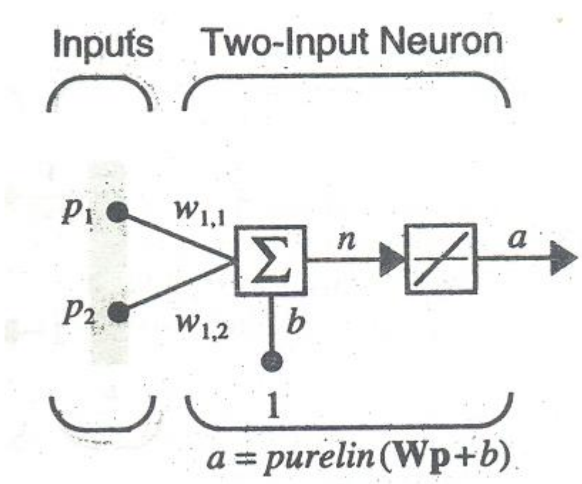
$$a_i = hardlim\ (n_i) = hardlim(W_i^T p + b_i). \qquad (44)$$

Recall that the *hardlim* transfer function (shown at left) is defined as:

Therefore, if the inner product of the ith row of the weight matrix with the input vector is greater than or equal to $-b_i$, the output will be 1, otherwise the (output will be 0. *Thus each neuron in the network divides the input space into two regions.* It is useful to investigate the boundaries between these regions. We will begin with the simple case of a single-neuron perceptron with two inputs.

Single-Neuron Perceptron

Let's consider a two-input perceptron with one neuron, as shown in figure below.



The output of this network is determined by

$$a = hardlim(n) = hardlim(Wp + b)$$

$$= hardlim(W_1^T p + \mathrm{b}) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b) \qquad (45)$$

The *decision boundary* is determined by the input vectors for which the net input n is zero:

$$n = W_1^T p + b = w_{1,1}p_1 + w_{1,2}p_2 + b = 0. \tag{46}$$

To make the example more concrete, let's assign the following values for the weights and bias:

$$w_{1,1} = 1, w_{1,2} = 1, b = -1. \tag{47}$$

The decision boundary is then

$$n = W_1^T p + b = w_{1,1}p_1 + w_{1,2}p_2 + b = p_1 + p_2 - 1 = 0 \tag{48}$$

This defines a line in the input space. On one side of the line the network output will be 0; on the line and on the other side of the line the output will be 1. To draw the line, we can find the points where it intersects the p_1 and p_2 axes. To find the p_2 intercept set $p_1 = 0$:

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{-1}{1} = 1 \; if \; p_1 = 0. \tag{49}$$

To find the p_1 intercept, set $p_2 = 0$;

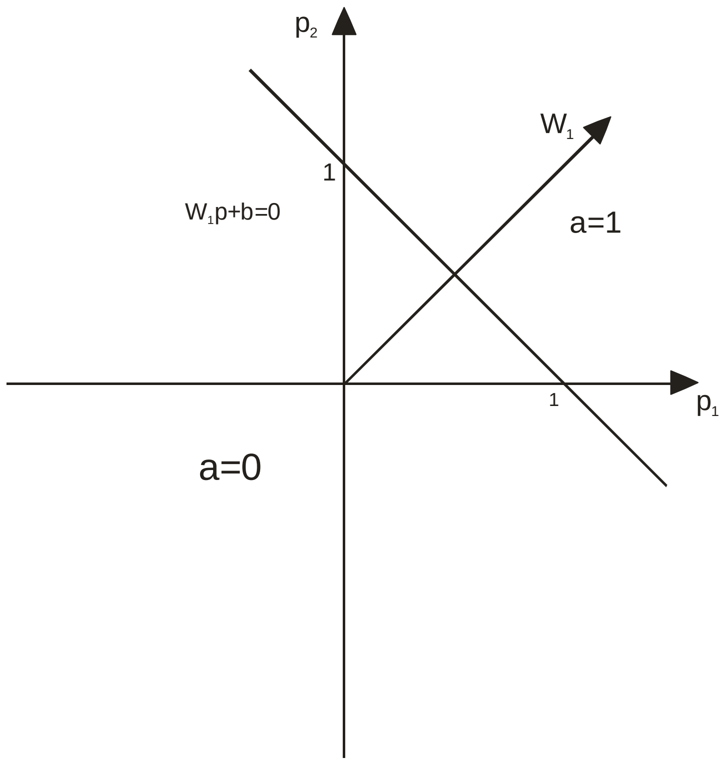
$$p_1 = -\frac{b}{w_{1,1}} = -\frac{-1}{1} = 1 \; if \; p_2 = 0. \tag{50}$$

The resulting decision boundary is illustrated in figure 4.3.

To find out which side of the boundary corresponds to an output of 1, we just need to test one point. For the input $p = [2 \; 0]^T$, the network output will be

$$a = hardlim(W_1^T p + b) = hardlim\left(\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1\right) = 1. \tag{51}$$

Therefore, the network output will be 1 for the region above and to the right of the decision boundary. This region is indicated by the shaded area in the following figure.



We can also find the decision boundary graphically. The first step is to note that the boundary is always orthogonal to W_1, as illustrated in the adjacent figures. The boundary is defined by

$$W_1^T \text{p+b=0} \tag{52}$$

For all points on the boundary, the inner product of the input vector with the weight vector is the same. This implies that these input vectors will all have the same projection onto the weight vector, so they must lie on a line orthogonal to the weight vector. In addition, any vector in the shaded region of the above figure will have an inner product greater than *-b,* and vectors in the unshaded region will have inner products less than *-b*. Therefore the weight vector $_i$w will always point toward the region where the neuron output is 1.

After we have selected a weight vector with the correct angular orientation, the bias value can be computed by selecting a point on the boundary and satisfying Eq. (52).

Let's apply some of these concepts to the design of a perceptron network to implement a simple logic function: the AND gate. The input/target pairs for the AND gate are

$$\left\{p_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}\left\{p_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\}\left\{p_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\}\left\{p_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}.$$

The figure to the left illustrated the problem graphically. It displays the input space, with each input vector labeled according to its target. The dark circles ● indicate that the target is 1, and the light circles ○ indicate that the target is 0.

The first step of the design is to select a decision boundary. We want to have a line that separates the dark circles and the light circles. There are an infinite number of solutions to this problem. It seems reasonable to choose the line that falls "halfway" between the two categories of inputs, as shown in the adjacent figure.

Next we want to choose a weight vector that is orthogonal ~ the decision boundary. The weight vector can be any length, so there are infinite possibilities.

One choice is

$$W_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \tag{53}$$

as displayed in the figure to the left.

Finally, we need to find the bias, b. We can do this by picking a point on the decision boundary and satisfying Eq. (4.15). If we use p=[1.5 0] we find

$$W_1^T p + b = [2 \quad 2] \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} + b = 3 + b = 0 \implies b = -3 \tag{54}$$

We can now test the network on one of the input/target pairs. If we apply p_2 to the network, the output will be

$$a = hardlim(W_1^T p_1 + \mathrm{b}) = hardlim\left([2 \quad 2] \begin{bmatrix} 0 \\ 1 \end{bmatrix} - 3\right) = hardlim(-1) = 0. \tag{55}$$

which is equal to the target t_2. Verify for yourself that all inputs are correctly classified.

Multiple-Neuron Perceptron

Note that for perceptrons with multiple neurons, there will be one decision boundary for each neuron. The decision boundary for neuron i will be defined by

$$W_i^T + b_i = 0 \tag{56}$$

A single-neuron perceptron can classify input vectors into two categories, since its output can be either 0 or 1. A multiple-neuron perceptron can classify inputs into many categories. Each category is represented by a different output vector. Since each element of the output vector can be either 0 or 1, there are a total of 2^S possible categories, where S is the number of neurons.

Perceptron Learning Rule

Now that we have examined the performance of perceptron networks, we are in a position to introduce the perceptron learning rule. This learning rule is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

$$\{p_1, t_1\}, \{p_2, t_2\}, \cdots, \{p_Q, t_Q\}, \qquad (57)$$

where 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 learning rule then adjusts the weights and biases of the network in order to move the network output closer to the target.

Test problem

In our presentation of the perceptron learning rule we will begin with a simple test problem and will experiment with possible rules to develop some intuition about how the rule should work. The input/target pairs for our test problem are

$$\left\{p_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\}\left\{p_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\}\left\{p_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}.$$

The problem is displayed graphically in the adjacent figure, where the two input vectors whose target is 0 are represented with a light circle ○, and the vector whose target is 1 is represented with a dark circle ●. This is a very simple problem, and we could almost obtain a solution by inspection. This simplicity will help us gain some intuitive understanding of the basic concepts of the perceptron learning rule.

The network for this problem should have two-inputs and one output. To simplify our development of the learning rule, we will begin with a network without a bias. The network will then have just two parameters $w_{1,1}$ and $w_{1,2}$. By removing the bias we are left with a network whose decision boundary must pass through the origin. We need to be sure that this network is still able to solve the test problem. There must be an allowable decision boundary that can separate vectors p_2 and p_3 from the vector p_1. The figure to the left illustrated that there are indeed an infinite number of such boundaries.

The adjacent figure shows the weight vectors that correspond to the allowable decision boundaries. (Recall that the weight vector is orthogonal to the decision boundary.) We would like a learning rule that will find a weight vector that points in one of these directions. Remember that the length of the weight vector does not matter; only its direction is important.

Constructing Learning Rules

Training begins by assigning some initial values for the network parameters. In this case we are training a two-input/single-output network without a bias, so we only have to initialize its two weights. Here we set the elements of the weight vector, ${}_1w$, to the following randomly generated values:

$$W_1^T = [1.0 \quad -0.8]. \qquad (58)$$

We will now begin presenting the input vectors to the network. We begin with p_1:

$$a = hardlim(W_1^T p_1) = hardlim\left([1.0 \quad -0.8]\begin{bmatrix}1\\2\end{bmatrix}\right) = hardlim(-0.6) = 0 \qquad (59)$$

The network has, not returned the correct value. The network output is 0, while the target response, t_1, is 1.

We can see what happened by looking at the adjacent diagram. The initial weight vector results in a decision boundary that incorrectly classifies the vector p_1. We need to alter the weight vector so that it points more toward p_1, so that in the future it has a better chance of classifying it correctly.

One approach would be to set I w equal to p_1. This is simple and would ensure that p_1 was classified properly in the future. Unfortunately, it is easy to construct a problem for which this rule cannot find a solution. The diagram to the lower left shows a problem that cannot be solved with the weight vector pointing directly at either of the two class 1 vectors. If we apply the rule I $_1w$ = P every time one of these vectors is misclassified, the network's weights will simply oscillate back and forth and will never find a solution.

Another possibility would be to add p_1 to $_1w$. Adding to p_1 to $_1w$ would make $_1w$ point more in the direction of p_1. Repeated presentations of p_1 would cause the direction of $_1w$ to asymptotically approach the direction of p_1.This rule can be stated: If *t=1* and *a=0*, then

$$_1w^{new}={_1w^{old}}+p. \tag{60}$$

Applying this rule to our test problem results in new values for $_1w$:

$$_1w^{new}={_1w^{old}}+p_1=\begin{bmatrix}1.0\\-0.8\end{bmatrix}+\begin{bmatrix}1\\2\end{bmatrix}=\begin{bmatrix}2.0\\1.2\end{bmatrix} \tag{61}$$

This operation is illustrated in the adjacent figure.

We now move on to the next input vector and will continue making changes to the weights and cycling through the inputs until they are all classified correctly.

The next input vector is p_2. When it is presented to the network we find:

$$a = hardlim({_1w^T}p_2)= hardlim\left(\begin{bmatrix}2.0 & 1.2\end{bmatrix}\begin{bmatrix}-1\\2\end{bmatrix}\right) = hardlim(0.4) = 1 \tag{62}$$

The target t_2 associated with p_2 is 0 and the output a is 1. A class 0 vector is misclassified as a 1.

Since we would now like to move the weight vector $_1w$ away from the input, we can simply change the addition in Eq. (60) to subtraction:

$$\text{If } t=0 \text{ and } a=1, \text{ then } {_1w^{new}}={_1w^{old}}-p. \tag{63}$$

If we apply this to the test problem we find:

$$_1w^{new}={_1w^{old}}-p_2=\begin{bmatrix}2.0\\1.2\end{bmatrix}-\begin{bmatrix}-1\\2\end{bmatrix}=\begin{bmatrix}3.0\\-0.8\end{bmatrix} \tag{64}$$

which is illustrated in the adjacent figure.

Now we present the third vector p_3:

$$a = hardlim(W_1^T p_3) = hardlim\left(\begin{bmatrix}3.0 & -0.8\end{bmatrix}\begin{bmatrix}0\\-1\end{bmatrix}\right) = hardlim(0.8) = 1 \qquad (65)$$

The current ${}_1w$ results in a decision boundary that misclassifies p_3. This is a situation for which we already have a rule, so ${}_1w$ will be updated again, according to Eq. (63):

$${}_1w^{new} = {}_1w^{old} - p_3 = \begin{bmatrix}3.0\\-0.8\end{bmatrix} - \begin{bmatrix}0\\-1\end{bmatrix} = \begin{bmatrix}3.0\\0.2\end{bmatrix} \qquad (66)$$

The diagram to the left shows that the perceptron has finally learned to classify the three vectors properly. If we present any of the input vectors to the neuron, it will output the correct class for that input vector.

This brings us to our third and final rule: if it works, don't fix it.

$$\text{If } t=a, \text{ then } {}_1w^{new} = {}_1w^{old}. \qquad (67)$$

Here are the three rules, which cover all possible combinations of output and target values:

$$\begin{gathered}\text{If } t=1 \text{ and } a=0, \text{ then } {}_1w^{new} = {}_1w^{old} + p.\\ \text{If } t=0 \text{ and } a=1, \text{ then } {}_1w^{new} = {}_1w^{old} - p.\\ \text{If } t=a, \text{ then } {}_1w^{new} = {}_1w^{old}.\end{gathered} \qquad (68)$$

Unified Learning Rule

The three rules in Eq. (68) can be rewritten as a single expression. First we will define a new variable, the perceptron error *e:*

$$e = t - a. \qquad (69)$$

We can now rewrite the three rules of Eq. (4.31) as:

$$\begin{gathered}\text{If } e=1, \text{ then } {}_1w^{new} = {}_1w^{old} + p.\\ \text{If } e=-1, \text{ then } {}_1w^{new} = {}_1w^{old} - p.\\ \text{If } e=0, \text{ then } {}_1w^{new} = {}_1w^{old}.\end{gathered} \qquad (70)$$

Looking carefully at the first two rules in Eq. (70) we can see that the sign of p is the same as the sign on the error, *e*. Furthermore, the absence of p in the third rule corresponds to an *e* of 0.

Thus, we can unify the three rules into a single expression:

$${}_1w^{new} = {}_1w^{old} + ep = {}_1w^{old} + (t-a)p. \qquad (71)$$

This rule can be extended to train the bias by noting that a bias is simply a weight whose input is always 1. We can thus replace the input p in Eq. (71) with the input to the bias, which is 1. The result is the perceptron rule for a bias:

$$b^{new} = b^{old} + e \qquad (72)$$

Training Multiple-Neuron Perceptrons

The perceptron rule, as given by Eq. (73) and Eq. (74), updates the weight vector of a single neuron perceptron. We can generalize this rule for the multiple-neuron perceptron of Figure 4.1 as follows. To update the ith row of the weight matrix use:

$$_iw^{new} = {_iw^{old}} + e_i p. \tag{73}$$

To update the ith element of the bias vector use:

$$b_i^{new} = b_i^{old} + e_i. \tag{74}$$

The *perceptron rule* can be written conveniently in matrix notation:

$$W^{new} = W^{old} + ep^T, \tag{75}$$

and

$$b^{new} = b^{old} + e. \tag{76}$$

To test the perceptron learning rule, consider again the apple/orange recognition problem of Chapter 3. The input/output prototype vectors will be

$$\left\{p_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = [0]\right\} \quad \left\{p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = [1]\right\}. \tag{77}$$

(Note that we are using 0 as the target output for the orange pattern, p_1 instead of -1, as was used in Perceptron. This is because we are using the *hardlim* transfer function, instead of *hardlims* .)

Typically the weights and biases are initialized to small random numbers. Suppose that here we start with the initial weight matrix and bias:

$$W=[0.5 \ \ -1 \ \ -0.5], \ \ b=0.5. \tag{78}$$

The first step is to apply the first input vector, p_1, to the network:

$$a = hardlim(Wp_1 + b) = hardlim\left([0.5 \quad -1 \quad -0.5]\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5\right) \tag{79}$$

$$= hardlim(2.5) = 1$$

Then we calculate the error:

$$e = t_1 - a = 0 - 1 = -1. \tag{80}$$

The weight update is

$$W^{new} = W^{old} + ep^T = [0.5 \quad -1 \quad -0.5] + (-1)[1 \quad -1 \quad -1] = [-0.5 \quad 0 \quad 0.5.] \tag{81}$$

The bias update is

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5\,. \tag{82}$$

This completes the first iteration.

The second iteration of the perceptron rule is:

$$a = hardlim(Wp_2 + b) = hardlim\left([-0.5 \quad -1 \quad 0.5]\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5)\right) \tag{83}$$

$$= hardlim(-0.5) = 0$$

$$e = t_2 - a = 1 - 0 = 1. \tag{84}$$

$$W^{new} = W^{old} + ep^T = [-0.5 \quad 0 \quad 0.5] + 1[1 \quad 1 \quad -1] = [0.5 \quad 1 \quad -0.5] \tag{85}$$

$$b^{new} = b^{old} + e = -0.5 + (1) = \ 0.5\,. \tag{86}$$

The third iteration begins again with the first input vector:

$$a = hardlim(Wp_1 + b) = hardlim\left([0.5 \quad 1 \quad 0.5]\begin{bmatrix}1\\-1\\-1\end{bmatrix} + 0.5\right) \tag{87}$$

$$= hardlim(0.5) = 1$$

$$e = t_1 - a = 0 - 1 = -1. \tag{88}$$

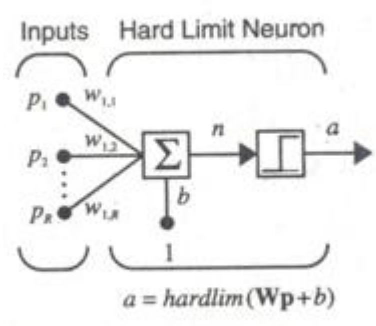
$$W^{new} = W^{old} + ep^T = [0.5 \quad 1 \quad -0.5] + (-1)[1 \quad 1 \quad -1] = [-0.5 \quad 2 \quad 0.5] \tag{89}$$

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5\,. \tag{90}$$

If you continue with the iterations you will find that both input vectors will now be correctly classified. The algorithm has converged to a solution. Note that the final decision boundary is not the same as the one we developed in Perceptron, although both boundaries correctly classify the two input vectors.

Proof of Convergence

Although the perceptron learning rule is simple, it is quite powerful. In fact, it can be shown that the rule will always converge to weights that accomplish the desired classification (assuming that such weights exist). In this section we will present a proof of convergence for the perceptron learning rule for the single-neuron perceptron shown in figure given below.



The output of this perceptron is obtained from

$$a = hardlilm(W_1^T p + b). \tag{91}$$

The network is provided with the following examples of proper network behavior:

$$\{p_1, t_1\}, \{p_2, t_2\}, \cdots, \{p_Q, t_Q\}, \tag{92}$$

where each target output, t_q, either o or 1.

Notation

To conveniently present the proof we will first introduce some new notation. We will combine the weight matrix and the bias into a single vector.

We will also augment the input vectors with a 1, corresponding to the bias input:

$$z_q = \begin{bmatrix} p_q \\ 1 \end{bmatrix}. \tag{93}$$

Now we can express the net input to the neuron as follows:

$$\text{n}=W_1^T p + b = x^T z. \tag{94}$$

The perceptron learning rule for a single-neuron perceptron (Eq. (71) and Eq. (72)) can be written

$$X^{new} = X^{old} + ez. \tag{95}$$

The error *e* can be either 1, -l or 0. If *e* = 0, then no change is made to the weights. If *e* = 1, then the input vector is added to the weight vector. If *e* = -1, then the negative of the input vector is added to the weight vector. If we count only those iterations for which the weight vector is changed, the learning rule becomes

$$x(k) = x(k-1) + z'(k-1), \tag{96}$$

where $z'(k-1)$ is the appropriate member of the sets.

$$\{z_1, z_2, \cdots, z_Q, -z_1, -z_2, \cdots, -z_Q\}. \tag{97}$$

We will assume that a weight vector exists that can correctly categorize all Q input vectors. This solution will be denoted x*. For this weight vector we will assume that

$$x^{*^T} z_q > \delta > 0 \text{ if } t_q = 1, \tag{98}$$

and

$$x^{*^T} z_q < -\delta < 0 \text{ if } t_q = 0, \tag{99}$$

Proof:

We are now ready to begin the proof of the perceptron convergence theorem. The objective of the proof is to find upper and lower bounds on the length of the weight vector at each stage of the algorithm. Assume that the algorithm is initialized with the zero weight vector: x (0) = 0. (This does not affect the generality of our argument.) Then, after *k* iterations (changes to the weight vector), we find from Eq. (96):

$$x(k) = z'(0) + z'(1) + \cdots + z'(k-1). \tag{100}$$

If we take the inner product of the solution weight vector with the weight vector at iteration k we obtain

$$x^{*^T} x(k) = x^{*^T} z'(0) + x^{*^T} z'(1) + \cdots + x^{*^T} z'(k-1). \tag{101}$$

From Eq. (97)-Eq. (99) we can show that

$$x^{*T} z'(i) > \delta. \tag{102}$$

Therefore

$$x^{*T} x(k) > k\delta. \tag{103}$$

From the Cauchy-Schwartz inequality

$$\left(x^{*T} x(k)\right)^2 \leq \|x^*\|^2 \|x(k)\|^2, \tag{104}$$

where

$$\|x\|^2 = x^T x. \tag{105}$$

If we combine Eq. (103) and Eq. (104) we can put a lower bound on the squared length of the weight vector at iteration *k:*

$$\|x(k)\|^2 \geq \frac{\left(x^{*T} x(k)\right)^2}{\|x^*\|^2} > \frac{(k\delta)^2}{\|x^*\|^2}. \tag{106}$$

Next we want to find an upper bound for the length of the weight vector. We begin by finding the change in the length at iteration *k:*

$$\begin{aligned} \|x(k)\|^2 &= x^T(k)x(k) \\ &= [x(k-1) + z'(k-1)]^T [x(k-1) + z'(k-1)] \\ &= x^T(k-1)x(k-1) + 2x^T(k-1)z'(k-1) + z'^T(k-1)z'(k-1). \end{aligned} \tag{107}$$

Note that

$$x^T(k-1)z'(k-1) \leq 0, \tag{108}$$

since the weights would not be updated unless the previous input vector had been misclassified. Now Eq. (107) can be simplified to

$$\|x(k)\|^2 \leq \|x(k-1)\|^2 + \|z'(k-1)\|^2. \tag{109}$$

We can repeat this process for $\|x(k-1)\|^2, \|x(k-2)\|^2$, etc, to obtain

$$\|x(k)\|^2 \leq \|z'(0)\|^2 + \cdots + \|z'(k-1)\|^2. \tag{110}$$

If $\prod = \max\{\|z'(i)\|^2\}$, this upper bound can be simplified to

$$\|x(k)\|^2 \leq k \prod. \tag{111}$$

We now have an upper bound (Eq. (111)) and a lower, bound (Eq. (106)) on the squared length of the weight vector at iteration *k.* If we combine the two inequalities we find

$$k\Pi \geq \|x(k)\|^2 > \frac{(k\delta)^2}{\|x^*\|^2} \text{ or } k < \frac{\Pi \|x^*\|^2}{\delta^2}. \tag{112}$$

Because *k* has an upper bound, this means that the weights will only be changed a finite number of times. Therefore, the perceptron learning rule will converge in a finite number of iterations.

The maximum number of iterations (changes to the weight vector) is inversely related to the square of 0. This parameter is a measure of how close the solution decision boundary is to the input patterns. This means that if the input classes are difficult to separate (are close to the decision boundary) it will take many iterations for the algorithm to converge.

Note that there are only three key assumptions required for the proof:

1. A solution to the problem exists, so that Eq. (102) is satisfied

2. The weights are only updated when the input vector is misclassified; therefore Eq. (108) is satisfied.
3. AD upper bound, n, exists for the length of the input vectors

Because of the generality of the proof, there are many variations of the perceptron learning rule that can also be shown to converge.

Limitations

The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists. This brings us to an important question. What problems can a perceptrgn solve? Recall that a single-neuron perceptron is able to divide the input space into two regions. The boundary between the regions is defined by the equation

$$w_1^T p + b = 0. \qquad (113)$$

This is a linear boundary (hyperplane). The perceptron can be used to classify input vectors that can be separated by a linear boundary. We call such vectors *linearly separable.* The logical AND gate example illustrates a two-dimensional example of a linearly separable problem. The apple/orange recognition problem of Hopfield network was a three-dimensional example.

Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. The input/target pairs for the XOR gate are

$$\left\{p_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{p_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\} \left\{p_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\} \left\{p_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}.$$

This problem is illustrated graphically on the left side, which also shows two other linearly inseparable problems. Try drawing a straight line between the vectors with targets of 1 and those with targets of 0 in any of the diagrams.

It was the inability of the basic perceptron to solve such simple problems that led, in part, to a reduction in interest in neural network research during the 1970s. Rosenblatt had investigated more complex networks, which he felt would overcome the limitations of the basic perceptron, but he was never able to effectively extend the perceptron rule to such networks. In Backpropagation we will introduce multilayer perceptrons, which can solve arbitrary classification problems, and will describe the backpropagation algorithm, which can be used to train them.