Chapter One
Neural networks—an overview

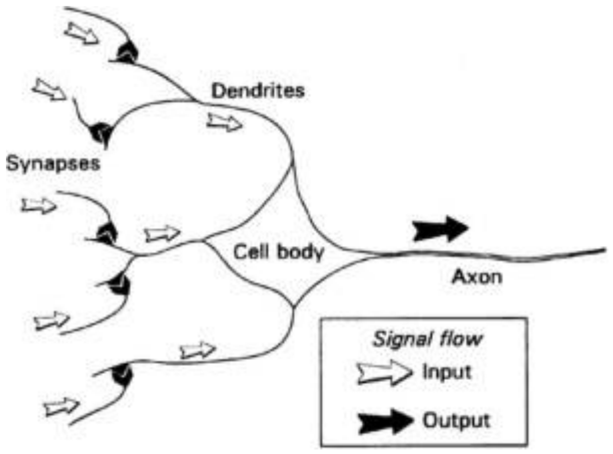


Figure 1.1 Essential components of a neuron shown in stylized form.

The term "Neural networks" is a very evocative one. It suggests machines that are something like brains and is potentially laden with the science fiction connotations of the Frankenstein mythos. One of the main tasks of this book is to demystify neural networks and show how, while they indeed have something to do with brains, their study also makes contact with other branches of science, engineering and mathematics. The aim is to do this in as non-technical a way as possible, although some mathematical notation is essential for specifying certain rules, procedures and structures quantitatively. Nevertheless, all symbols and expressions will be explained as they arise so that, hopefully, these should not get in the way of the essentials: that is, concepts and ideas that may be described in words.

This chapter is intended for orientation. We attempt to give simple descriptions of what networks are and why we might study them. In this way, we have something in mind right from the start, although the whole of this book is, of course, devoted to answering these questions in full.

1.1
What are neural networks?

Let us commence with a provisional definition of what is meant by a "neural network" and follow with simple, working explanations of some of the key terms in the definition.

> A neural network is an interconnected assembly of simple processing elements, *units* or *nodes,* whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the interunit connection strengths, or *weights,* obtained by a process of adaptation to, or *learning* from, a set of training patterns.

To flesh this out a little we first take a quick look at some basic neurobiology. The human brain consists of an estimated 10^{11} (100 billion) nerve cells or *neurons,* a highly stylized example of which is shown in Figure 1.1. Neurons communicate via electrical signals that are short-lived impulses or "spikes" in the voltage of the cell wall or *membrane.* The interneuron connections are mediated by electrochemical junctions called *synapses,* which are located on branches of the cell referred to as *dendrites*. Each neuron typically receives many thousands of connections from other neurons and is therefore constantly receiving a multitude of incoming signals, which eventually reach the cell body. Here, they are integrated or summed together

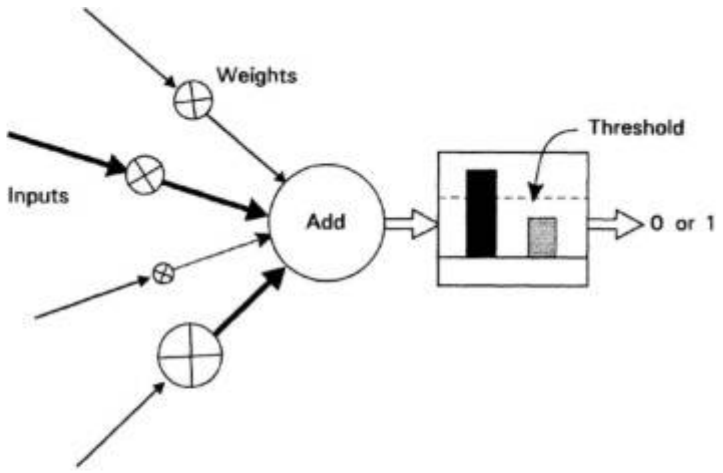


Figure 1.2 Simple artificial neuron.

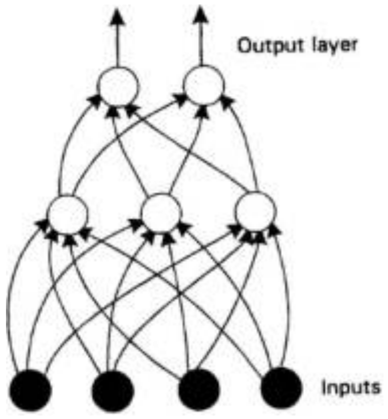


Figure 1.3 Simple example of neural network.

in some way and, roughly speaking, if the resulting signal exceeds some threshold then the neuron will "fire" or generate a voltage impulse in response. This is then transmitted to other neurons via a branching fibre known as the *axon*.

In determining whether an impulse should be produced or not, some incoming signals produce an inhibitory effect and tend to prevent firing, while others are excitatory and promote impulse generation. The distinctive processing ability of each neuron is then supposed to reside in the type—excitatory or inhibitory— and strength of its synaptic connections with other neurons.

It is this architecture and style of processing that we hope to incorporate in neural networks and, because of the emphasis on the importance of the interneuron connections, this type of system is sometimes referred to as being *connectionist* and the study of this general approach as *connectionism*. This terminology is often the one encountered for neural networks in the context of psychologically inspired models of human cognitive function. However, we will use it quite generally to refer to neural networks without reference to any particular field of application.

The artificial equivalents of biological neurons are the nodes or units in our preliminary definition and a prototypical example is shown in Figure 1.2. Synapses are modelled by a single number or *weight* so that each input is multiplied by a weight before being sent to the equivalent of the cell body. Here, the weighted signals are summed together by simple arithmetic addition to supply a node *activation*. In the type of node shown in Figure 1.2—the so-called *threshold logic unit* (TLU)—the activation is then compared with a threshold; if the activation exceeds the threshold, the unit produces a high-valued output (conventionally "1"), otherwise it outputs zero. In the figure, the size of signals is represented by the width of their corresponding arrows, weights are shown by multiplication symbols in circles, and their values are supposed to be proportional to the symbol's size; only positive weights have been used. The TLU is the simplest (and historically the earliest (McCulloch & Pitts 1943)) model of an artificial neuron.

The term "network" will be used to refer to any system of artificial neurons. This may range from something as simple as a single node to a large collection of nodes in which each one is connected to every other node in the net. One type of network is shown in Figure 1.3. Each node is now shown by only a circle but weights are implicit on all connections. The nodes are arranged in a layered structure in which each signal emanates from an input and passes via two nodes before reaching an output beyond which it is no longer transformed. This *feedforward* structure is only one of several available and is typically used to place an input pattern into one of several classes according to the resulting pattern of outputs. For example, if the input consists of an encoding of the patterns of light and dark in an image of handwritten letters, the output layer (topmost in the figure) may contain 26 nodes—one for each letter of the alphabet—to flag which letter class the input character is from. This would be done by allocating one output node per class and requiring that only one such node fires whenever a pattern of the corresponding class is supplied at the input.

So much for the basic structural elements and their operation. Returning to our working definition, notice the emphasis on learning from experience. In real neurons the synaptic strengths may, under certain circumstances, be modified so that the behaviour of each neuron can change or adapt to its particular stimulus input. In artificial neurons the equivalent of this is the modification of the weight values. In terms of processing information, there are no computer programs here —the "knowledge" the network has is supposed to be stored in its weights, which evolve by a process of adaptation to stimulus from a set of pattern examples. In one training paradigm called *supervised learning,* used in conjunction with nets of the type shown in Figure 1.3, an input pattern is presented to the net and its response then compared with a target output. In terms of our previous letter recognition example, an "A", say, may be input and the network output compared with the classification code for A. The difference between the two patterns of output then determines how the weights are altered. Each particular recipe for change constitutes a *learning rule,* details of which form a substantial part of subsequent chapters. When the required weight updates have been made another pattern is presented, the output compared with the target, and new changes made. This sequence of events is repeated iteratively many times until (hopefully) the network's behaviour converges so that its response to each pattern is close to the corresponding target. The process as a whole, including any ordering of pattern presentation, criteria for terminating the process, etc., constitutes the *training algorithm*.

What happens if, after training, we present the network with a pattern it hasn't seen before? If the net has learned the underlying structure of the problem domain then it should classify the unseen pattern correctly and the net is said to *generalize* well. If the net does not have this property it is little more than a classification lookup table for the training set and is of little practical use. Good generalization is therefore one of the key properties of neural networks.

1.2 Why study neural networks?

This question is pertinent here because, depending on one's motive, the study of connectionism can take place from differing perspectives. It also helps to know what questions we are trying to answer in order to avoid the kind of religious wars that sometimes break out when the words "connectionism" or "neural network" are mentioned.

Neural networks are often used for statistical analysis and data modelling, in which their role is perceived as an alternative to standard nonlinear regression or cluster analysis techniques (Cheng & Titterington 1994). Thus, they are typically used in problems that may be couched in terms of classification, or forecasting. Some examples include image and speech recognition, textual character recognition, and domains of human expertise such as medical diagnosis, geological survey for oil, and financial market indicator prediction. This type of problem also falls within the domain of classical artificial intelligence (AI) so that engineers and computer scientists see neural nets as offering a style of *parallel distributed computing,* thereby providing an alternative to the conventional algorithmic techniques that have dominated in machine intelligence. This is a theme pursued further in the final chapter but, by way of a brief explanation of this term now, the parallelism refers to the fact that each node is conceived of as operating independently and concurrently (in parallel with) the others, and the "knowledge" in the network is distributed over the entire set of weights, rather than focused in a few memory locations as in a conventional computer. The practitioners in this area do not concern themselves with biological realism and are often motivated by the ease of implementing solutions in digital hardware or the efficiency and accuracy of particular techniques. Haykin (1994) gives a comprehensive survey of many neural network techniques from an engineering perspective.

Neuroscientists and psychologists are interested in nets as computational models of the animal brain developed by abstracting what are believed to be those properties of real nervous tissue that are essential for information processing. The artificial neurons that connectionist models use are often extremely simplified versions of their biological counterparts and many neuroscientists are sceptical about the ultimate power of these impoverished models, insisting that more detail is necessary to explain the brain's function. Only time will tell but, by drawing on knowledge about how real neurons are interconnected as local "circuits", substantial inroads have been made in modelling brain functionality. A good introduction to this programme of *computational neuroscience* is given by Churchland & Sejnowski (1992).

Finally, physicists and mathematicians are drawn to the study of networks from an interest in nonlinear dynamical systems, statistical mechanics and automata theory.[1] It is the job of applied mathematicians to discover and formalize the properties of new systems using tools previously employed in other areas of science. For example, there are strong links between a certain type of net (the Hopfield net—see Ch. 7) and magnetic systems known as spin glasses. The full mathematical apparatus for exploring these links is developed (alongside a series of concise summaries) by Amit (1989).

All these groups are asking different questions: neuroscientists want to know how animal brains work, engineers and computer scientists want to build intelligent machines and mathematicians want to understand the fundamental properties of networks as complex systems. Another (perhaps the largest) group of people are to be found in a variety of industrial and commercial areas and use neural networks to model and analyze large, poorly understood datasets that arise naturally in their workplace. It is therefore important to understand an author's perspective when reading the literature. Their common focal point is, however, neural networks and is potentially the basis for close collaboration. For example, biologists can usefully learn from computer scientists what computations are necessary to enable animals to solve particular problems, while engineers can make use of the solutions nature has devised so that they may be applied in an act of "reverse engineering".

In the next chapter we look more closely at real neurons and how they may be modelled by their artificial counterparts. This approach allows subsequent development to be viewed from both the biological and engineering-oriented viewpoints.

1.3 Summary

Artificial neural networks may be thought of as simplified models of the networks of neurons that occur naturally in the animal brain. From the biological viewpoint the essential requirement for a neural network is that it should attempt to capture what we believe are the essential information processing features of the corresponding "real" network. For an engineer, this correspondence is not so important and the network offers an alternative form of parallel computing that might be more appropriate for solving the task in hand.

The simplest artificial neuron is the threshold logic unit or TLU. Its basic operation is to perform a weighted sum of its inputs and then output a "1" if this sum exceeds a threshold, and a "0" otherwise. The TLU is supposed to model the basic "integrate-and-fire" mechanism of real neurons.

1.4 Notes

1. It is not important that the reader be familiar with these areas. It suffices to understand that neural networks can be placed in relation to other areas studied by workers in these fields.

Chapter Two
Real and artificial neurons

The building blocks of artificial neural nets are artificial neurons. In this chapter we introduce some simple models for these, motivated by an attempt to capture the essential information processing ability of real, biological neurons. A description of this is therefore our starting point and, although our excursion into neurophysiology will be limited, some of the next section may appear factually rather dense on first contact. The reader is encouraged to review it several times to become familiar with the biological "jargon" and may benefit by first re-reading the précis of neuron function that was given in the previous chapter. In addition, it will help to refer to Figure 2.1 and the glossary at the end of the next section.

2.1
Real neurons: a review

Neurons are not only enormously complex but also vary considerably in the details of their structure and function. We will therefore describe typical properties enjoyed by a majority of neurons and make the usual working assumption of connectionism that these provide for the bulk of their computational ability. Readers interested in finding out more may consult one of the many texts in neurophysiology; Thompson (1993) provides a good introductory text, while more comprehensive accounts are given by Kandel et al. (1991) and Kuffler et al. (1984).

A stereotypical neuron is shown in Figure 2.1, which should be compared with the simplified diagram in Figure 1.1. The cell body or *soma* contains the usual subcellular components or *organelles* to be found in most cells throughout the body (nucleus, mitochondria, Golgi body, etc.) but these are not shown in the diagram. Instead we focus on what differentiates neurons from other cells allowing the neuron to function as a signal processing device. This ability stems largely from the properties of the neuron's surface covering or membrane, which supports a wide variety of electrochemical processes. Morphologically the main difference lies in the set of fibres that emanate from the cell body. One of these fibres—the axon—is responsible for transmitting signals to other neurons and may therefore be considered the neuron output. All other fibres are dendrites, which carry signals from other neurons to the cell body, thereby acting as neural inputs. Each neuron has only one axon but can have many dendrites. The latter often appear to have a highly branched structure and so we talk of dendritic *arbors*. The axon may, however, branch into a set of *collaterals* allowing contact to be made with many other neurons. With respect to a particular neuron, other neurons that supply input are said to be *afferent,* while the given neuron's axonal output, regarded as a projection to other cells, is referred to as an *efferent*. Afferent axons are said to *innervate* a particular neuron and make contact with dendrites at the junctions called *synapses*. Here, the extremity of the axon, or *axon terminal,* comes into close proximity with a small part of the dendritic surface—the *postsynaptic* membrane. There is a gap, the *synoptic cleft,* between the *presynaptic* axon terminal membrane and its postsynaptic counterpart, which is of the order of 20 nanometres (2×10^{-8}m) wide. Only a few synapses are shown in Figure 2.1 for the sake of clarity but the reader should imagine a profusion of these located over all dendrites and also, possibly, the cell body. The detailed synaptic structure is shown in schematic form as an inset in the figure.

So much for neural structure; how does it support signal processing? At equilibrium, the neural membrane works to maintain an electrical imbalance of negatively and positively charged *ions*. These are atoms or molecules that have a surfeit or deficit of electrons, where each of the latter carries a single negative charge. The net result is that there is a *potential difference* across the membrane with the inside being negatively *polarized* by approximately 70mV[1] with respect to the outside. Thus, if we could imagine applying a voltmeter to the membrane it would read 70mV, with the inside being more negative than the outside. The main point here is that a neural membrane can support electrical signals if its state of polarization or *membrane potential* is dynamically changed. To see this, consider the case of signal propagation along an axon as shown in Figure 2.2. Signals that are propagated along axons, or *action potentials,* all have the same characteristic shape, resembling sharp pulse-like spikes. Each graph shows a snapshot of the membrane potential along a segment of axon that is currently transmitting a single action potential, and the lower panel shows the situation at some later time with respect to the upper one. The ionic mechanisms at work to produce this process were first worked out by Hodgkin & Huxley (1952). It relies on the interplay between each of the ionic currents across the membrane and its mathematical description is complex. The details do

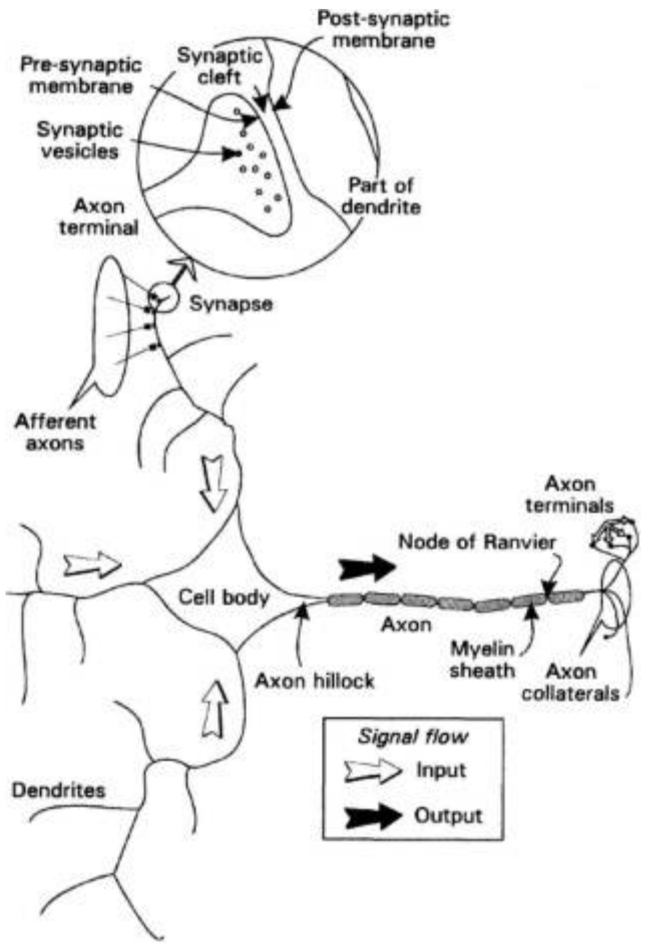


Figure 2.1 Biological neuron.

not concern us here, but this example serves to illustrate the kind of simplification we will use when we model using artificial neurons; real axons are subject to complex, nonlinear dynamics but will be modelled as a passive output “wire”. Many neurons have their axons sheathed in a fatty substance known as myelin, which serves to enable the more rapid conduction of action potentials. It is punctuated at approximately 1 mm intervals by small unmyelinated segments (nodes of Ranvier in Fig. 2.1), which act rather like “repeater stations” along a telephone cable.

We are now able to consider the passage of signals through a single neuron, starting with an action potential reaching an afferent axon terminal. These contain a chemical substance or *neurotransmitter* held within a large number of small *vesicles* (literally “little spheres”). On receipt of an action potential the vesicles migrate to the presynaptic membrane and release their neurotransmitter across the synaptic cleft. The transmitter then *binds* chemically with *receptor sites* at the postsynaptic membrane. This initiates an electrochemical process that changes the polarization state of the membrane local to the synapse. This *postsynaptic potential* (PSP) can serve either to *depolarize* the membrane from its negative resting state towards 0 volts, or to *hyperpolarize* the membrane to an even greater negative potential. As we shall see, neural signal production is encouraged by depolarization, so that PSPs which are positive are excitatory PSPs (EPSPs) while those which hyperpolarize the membrane are inhibitory (IPSPs). While action potentials all have the same characteristic signal profile and the same maximum value, PSPs can take on a continuous range of values depending on the efficiency of the synapse in utilizing the chemical transmitter to produce an electrical signal. The PSP spreads out from the synapse, travels along its associated dendrite towards the cell body and eventually reaches the *axon hillock*—the initial segment of the axon where it joins the soma. Concurrent with this are thousands of other synaptic events distributed over the neuron. These result in a plethora of PSPs, which are continually arriving at the axon hillock where they are summed together to produce a resultant membrane potential.

Each contributory PSP at the axon hillock exists for an extended time (order of milliseconds) before it eventually decays so that, if two PSPs arrive slightly out of synchrony, they may still interact in the summation process. On the other hand, suppose two synaptic events take place with one close to and another remote from the soma, by virtue of being at the end of a long dendritic branch. By the time the PSP from the distal (remote) synapse has reached the axon hillock, that originating close to the soma will have decayed. Thus, although the initiation of PSPs may take place in synchrony, they may not be effective in

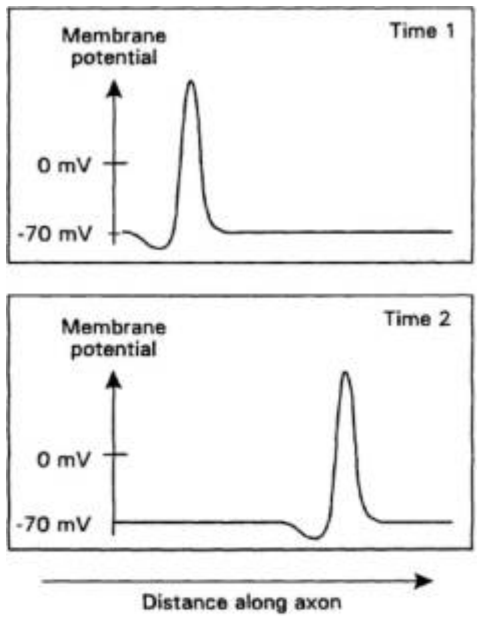


Figure 2.2 Action-potential propagation.

combining to generate action potentials. It is apparent, therefore, that a neuron sums or integrates its PSPs over both space *and* time. Substantial modelling effort—much of it pioneered by Rail (1957, 1959)—has gone into describing the conduction of PSPs along dendrites and their subsequent interaction although, as in the case of axons, connectionist models usually treat these as passive wires with no temporal characteristics.

The integrated PSP at the axon hillock will affect its membrane potential and, if this exceeds a certain threshold (typically about −50mV), an action potential is generated, which then propagates down the axon, along any collaterals, eventually reaching axon terminals resulting in a shower of synaptic events at neighbouring neurons “downstream” of our original cell. In reality the “threshold” is an emergent or meta-phenomenon resulting from the nonlinear nature of the Hodgkin-Huxley dynamics and, under certain conditions, it can be made to change. However, for many purposes it serves as a suitable high-level description of what actually occurs. After an action potential has been produced, the ionic metabolites used in its production have been depleted and there is a short *refractory period* during which, no matter what value the membrane potential takes, there can be no initiation of another action potential.

It is useful at this stage to summarize what we have learnt so far about the functionality of real neurons with an eye to the simplification required for modelling their artificial counterparts.

- Signals are transmitted between neurons by action potentials, which have a stereotypical profile and display an “all-or-nothing” character; there is no such thing as half an action potential.
- When an action potential impinges on a neuronal input (synapse) the effect is a PSP, which is variable or *graded* and depends on the physicochemical properties of the synapse.
- The PSPs may be excitatory or inhibitory.
- The PSPs are summed together at the axon hillock with the result expressed as its membrane potential.
- If this potential exceeds a threshold an action potential is initiated that proceeds along the axon.

Several things have been deliberately omitted here. First, the effect that synaptic structure can have on the value of the PSP. Factors that may play a role here include the type and availability of neurotransmitter, the postsynaptic receptors and synaptic geometry. Secondly, the spatio-temporal interdependencies of PSPs resulting from dendritic geometry whereby, for example, synapses that are remote from each other may not effectively combine. Finally, we have said nothing about the *dynamics* of action-potential generation and propagation. However, our summary will serve as a point of departure for defining the kind of artificial neurons described in this book. More biologically realistic models rely on solving Hodgkin-Huxley-type dynamics and modelling dendrites at the electrical circuit level; details of these methods can be found in the review compilation of Koch & Segev (1989).

2.1.1 Glossary of terms

Those terms in italics may be cross-referenced in this glossary.

action potential The stereotypical voltage spike that constitutes an active output from a neuron. They are propagated along the *axon* to other neurons.
afferent With respect to a particular neuron, an axon that impinges on (or *innervates*) that neuron.
arbor Usually used in the context of a dendritic arbor—the tree-like structure associated with dendritic branching.
axon The fibre that emanates from the neuron cell body or *soma* and that conducts *action potentials* to other neurons.
axon hillock The junction of the *axon* and cell body or *soma*. The place where *action potentials* are initiated if the *membrane potential* exceeds a threshold.
axon terminal An axon may branch into several *collaterals,* each terminating at an axon terminal, which constitutes the *presynaptic* component of a *synapse*.
chemical binding The process in which a *neurotransmitter* joins chemically with a *receptor site* thereby initiating a *PSP*.
collateral An axon may divide into many collateral branches allowing contact with many other neurons or many contacts with one neuron.
dendrite One of the branching fibres of a neuron, which convey input information via *PSPs.*
depolarization The *membrane potential* of the neuron has a negative resting or equilibrium value. Making this less negative leads to a depolarization. Sufficient depolarization at the *axon hillock* will give rise to an action potential.
efferent A neuron sends efferent axon *collaterals* to other neurons.
EPSP Excitatory Postsynaptic Potential. A *PSP* that acts to *depolarize* the neural membrane.
hyperpolarization The *membrane potential* of the neuron has a negative resting or equilibrium value. Making this more negative leads to a hyperpolarization and inhibits the action of *EPSPs,* which are trying to *depolarize* the membrane.
innervate Neuron *A* sending signals to neuron *B* is said to innervate neuron *B*.
IPSP Inhibitory Postsynaptic Potential. A *PSP* that acts to *hyperpolarize* the neural membrane.
membrane potential The voltage difference at any point across the neural membrane.
neurotransmitter The chemical substance that mediates synaptic activity by propagation across the *synaptic cleft*.
organelle Subcellular components that partake in metabolism, etc.
postsynaptic membrane That part of a *synapse* which is located on the *dendrite* and consists of the dendritic membrane together with *receptor sites*.
potential difference The voltage difference across the cell membrane.
presynaptic membrane That part of a *synapse* which is located on the *axon terminal*.
PSP Postsynaptic Potential. The change in *membrane potential* brought about by activity at a *synapse*.
receptor sites The sites on the *postsynaptic membrane* to which molecules of *neurotransmitter* bind. This binding initiates the generation of a *PSP*.
refractory period The shortest time interval between two *action potentials*.
soma The cell body.
synapse The site of physical and signal contact between neurons. On receipt of an *action potential* at the *axon terminal* of a *synapse, neurotransmitter* is released into the *synaptic cleft* and propagates to the *postsynaptic membrane*. There it undergoes *chemical binding* with *receptors,* which, in turn, initiates the production of a postsynaptic potential *(PSP)*.
synaptic cleft The gap between the pre- and postsynaptic membranes across which chemical *neurotransmitter* is propagated during synaptic action. vesicles The spherical containers in the *axon terminal* that contain *neurotransmitter*. On receipt of an action potential at the axon terminal, the vesicles release their neurotransmitter into the *synaptic cleft*.

2.2 Artificial neurons: the TLU

Our task is to try and model some of the ingredients in the list above. Our first attempt will result in the structure described informally in Section 1.1.

The "all-or-nothing" character of the action potential may be characterized by using a two-valued signal. Such signals are often referred to as *binary* or *Boolean*[2] and conventionally take the values "0" and "1". Thus, if we have a node receiving n input signals $x_1, x_2, \ldots, x_n$, then these may only take on the values "0" or "1". In line with the remarks of the previous chapter, the modulatory effect of each synapse is encapsulated by simply multiplying the incoming signal with a weight value, where excitatory and inhibitory actions are modelled using positive and negative values respectively. We therefore have n weights $w_1, w_2, \ldots, w_n$ and form the n products $w_1x_1, w_2x_2, \ldots, w_nx_n$. Each product is now the analogue of a PSP and may be negative or positive, depending on the sign of the weight. They should now be combined in a process which is supposed to emulate that taking place at the axon hillock. This will be done by simply adding them together to produce the activation a (corresponding to the axon-hillock membrane potential) so that

$$a = w_1x_1 + w_2x_2 + \cdots + w_nx_n \tag{2.1}$$

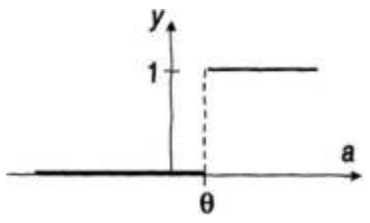


Figure 2.3 Activation-output threshold relation in graphical form.

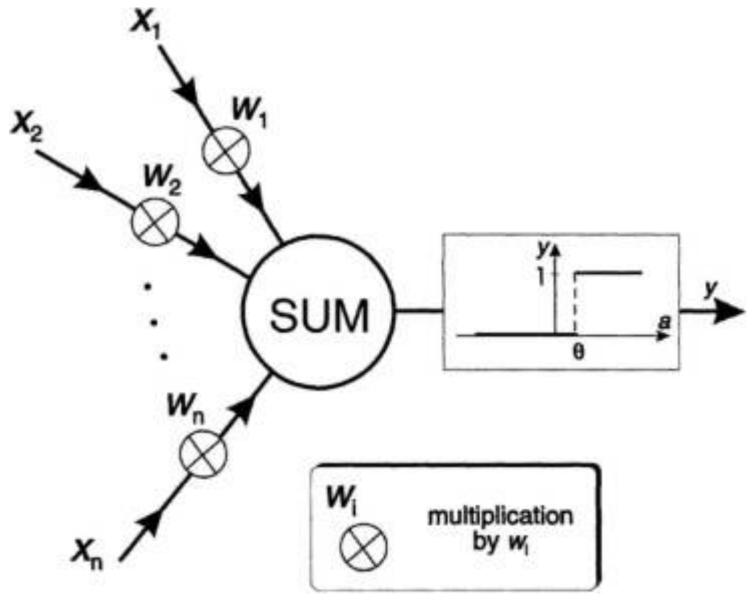


Figure 2.4 TLU.

As an example, consider a five-input unit with weights (0.5, 1.0, −1.0, −0.5, 1.2), that is w_1=0.5, w_2=1.0,…, w_5=1.2, and suppose this is presented with inputs (1, 1, 1, 0, 0) so that x_1=1, x_2=1,…, x_5=0. Using (2.1) the activation is given by

$$\begin{aligned} a &= (0.5 \times 1) + (1.0 \times 1) + (-1.0 \times 1) + (-0.5 \times 0) + (1.2 \times 0) \\ &= 0.5 \end{aligned}$$

To emulate the generation of action potentials we need a threshold value θ (Greek theta) such that, if the activation exceeds (or is equal to) θ then the node outputs a "1" (action potential), and if it is less than θ then it emits a "0". This may be represented graphically as shown in Figure 2.3 where the output has been designated the symbol y. This relation is sometimes called a *step function* or *hard-limiter* for obvious reasons. In our example, suppose that θ=0.2; then, since a>0.2 (recall a=0.5) the node's output y is 1. The entire node structure is shown in Figure 2.4 where the weights have been depicted by encircled multiplication signs. Unlike Figure 1.1, however, no effort has been made to show the size of the weights or signals. This type of artificial neuron is known as a threshold logic unit (TLU) and was originally proposed by McCulloch and Pitts (McCulloch & Pitts 1943).

It is more convenient to represent the TLU functionality in a symbolic rather than a graphical form. We already have one form for the activation as supplied by (2.1). However, this may be written more compactly using a notation that makes use of the way we have written the weights and inputs. First, a word on the notation is relevant here. The small numbers used in denoting the inputs and weights are referred to as *subscripts*. If we had written the numbers near the top (e.g. x^1) they would have been *superscripts* and, quite generally, they are called *indices* irrespective of their position. By writing the index symbolically (rather than numerically) we can refer to quantities generically so that x_i, for example, denotes the generic or ith input where it is assumed that i can be any integer between 1 and n. Similar remarks apply to the weights w_i. Using these ideas it is possible to represent (2.1) in a more compact form

$$a = \sum_{i=1}^{n} w_i x_i \qquad (2.2)$$

where Σ (upper case Greek sigma) denotes summation. The expressions above and below Σ denote the upper and lower limits of the summation and tell us that the index i runs from 1 to n. Sometimes the limits are omitted because they have been defined elsewhere and we simply indicate the summation index (in this case i) by writing it below the Σ.

The threshold relation for obtaining the output y may be written

$$y = \begin{cases} 1 & \text{if} \quad a \geq \theta \\ 0 & \text{if} \quad a < \theta \end{cases} \tag{2.3}$$

Notice that there is no mention of time in the TLU; the unit responds instantaneously to its input whereas real neurons integrate over time as well as space. The dendrites are represented (if one can call it a representation) by the passive connecting links between the weights and the summing operation. Action-potential generation is simply represented by the threshold function.

2.3 Resilience to noise and hardware failure

Even with this simple neuron model we can illustrate two of the general properties of neural networks. Consider a two-input TLU with weights (0, 1) and threshold 0.5. Its response to all four possible input sets is shown in Table 2.1.

Table 2.1 TLU with weights (0, 1) and threshold 0.5.

x_1	x_2	Activation	Output
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

Now suppose that our hardware which implements the TLU is faulty so that the weights are not held at their true values and are encoded instead as (0.2, 0.8). The revised TLU functionality is given in Table 2.2. Notice that, although the activation has changed, the output is the same as that for the original TLU. This is because changes in the activation, as long as they don't cross the threshold, produce no change in output. Thus, the threshold function doesn't care whether the activation is just below θ or is very much less than θ; it still outputs a 0. Similarly, it doesn't matter by how much the activation exceeds θ, the TLU always supplies a 1 as output.

Table 2.2 TLU with weights (0.2, 0.8) and threshold 0.5.

x_1	x_2	Activation	Output
0	0	0	0
0	1	0.8	1
1	0	0.2	0
1	1	1	1

This behaviour is characteristic of *nonlinear* systems. In a linear system, the output is proportionally related to the input: small/large changes in the input always produce corresponding small/large changes in the output. On the other hand, nonlinear relations do not obey a proportionality restraint so the *magnitude* of the change in output does not necessarily reflect that of the input. Thus, in our TLU example, the activation can change from 0 to 0.2 (a difference of 0.2) and make no difference to the output. If, however, it were to change from 0.49 to 0.51 (a difference of 0.02) the output would suddenly alter from 0 to 1.

We conclude from all this that TLUs are robust in the presence of hardware failure; if our hardware breaks down "slightly" the TLU may still function perfectly well as a result of its nonlinear functionality.

Table 2.3 TLU with degraded signal input.

x_1	x_2	Activation	Output
0.2	0.2	0.2	0
0.2	0.8	0.8	1
0.8	0.2	0.2	0
0.8	0.8	0.8	1

Suppose now that, instead of the weights being altered, the input signals have become degraded in some way, due to noise or a partial power loss, for example, so that what was previously "1" is now denoted by 0.8, and "0" becomes 0.2. The

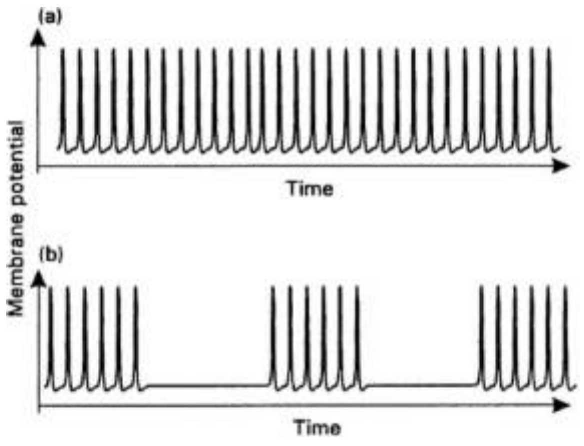


Figure 2.5 Neural firing patterns.

resulting TLU function is shown in Table 2.3. Once again the resulting TLU function is the same and a similar reasoning applies that involves the nonlinearity implied by the threshold. The conclusion is that the TLU is robust in the presence of noisy or corrupted signal inputs. The reader is invited to examine the case where both weights and signals have been degraded in the way indicated here. Of course, if we increase the amount by which the weights or signals have been changed too much, the TLU will eventually respond incorrectly. In a large network, as the degree of hardware and/or signal degradation increases, the number of TLU units giving incorrect results will gradually increase too. This process is called "graceful degradation" and should be compared with what happens in conventional computers where alteration to one component or loss of signal strength along one circuit board track can result in complete failure of the machine.

2.4 Non-binary signal communication

The signals dealt with so far (for both real and artificial neurons) have taken on only two values. In the case of real neurons these are the action-potential spiking voltage and the axon-membrane resting potential. For the TLUs they were conveniently labelled "1" and "0" respectively. Real neurons, however, are believed to encode their signal values in the *patterns* of action-potential firing rather than simply by the presence or absence of a single such pulse. Many characteristic patterns are observed (Conners & Gutnick 1990) of which two common examples are shown in Figure 2.5.

Part (a) shows a continuous stream of action-potential spikes while (b) shows a pattern in which a series of pulses is followed by a quiescent period, with this sequence repeating itself indefinitely. A continuous stream as in (a) can be characterized by the frequency of occurrence of action potential in pulses per second and it is tempting to suppose that this is, in fact, the code being signalled by the neuron. This was convincingly demonstrated by Hartline (1934, 1940) for the optic neurons of the Horseshoe crab *Limulus* in which he showed that the rate of firing increased with the visual stimulus intensity. Although many neural codes are available (Bullock et al. 1977) the frequency code appears to be used in many instances.

If f is the frequency of neural firing then we know that f is bounded below by zero and above by some maximum value f_{max}, which is governed by the duration of the interspike refractory period. There are now two ways we can code for f in our artificial neurons. First, we may simply extend the signal representation to a continuous range and directly represent f as our unit output. Such signals can certainly be handled at the input of the TLU, as we remarked in examining the effects of signal degradation. However, the use of a step function at the output limits the signals to be binary so that, when TLUs are connected in networks (and they are working properly), there is no possibility of continuously graded signals occurring. This may be overcome by "softening" the step function to a continuous "squashing" function so that the output y depends smoothly on the activation a. One convenient form for this is the *logistic sigmoid* (or sometimes simply "sigmoid") shown in Figure 2.6.

As a tends to large positive values the sigmoid tends to 1 but never actually reaches this value. Similarly it approaches—but never quite reaches—0 as a tends to large negative values. It is of no importance that the upper bound is not f_{max}, since we can simply multiply the sigmoid's value by f_{max} if we wish to interpret y as a real firing rate. The sigmoid is symmetric about the y-axis value of 0.5; the corresponding value of the activation may be thought of as a reinterpretation of the threshold and is denoted by θ. The sigmoid function is conventionally designated by the Greek lower case sigma, σ, and finds mathematical expression according to the relation

$$y = \sigma(a) \equiv \frac{1}{1+e^{-(a-\theta)/\rho}} \tag{2.4}$$

where $e \approx 2.7183$ is a mathematical constant[3], which, like π, has an infinite decimal expansion. The quantity ρ (Greek rho) determines the shape of the function, large values making the curve flatter while small values make the curve rise more steeply. In many texts, this parameter is omitted so that it is implicitly assigned the value 1. By making ρ progressively smaller we obtain functions that look ever closer to the hard-limiter used in the TLU so that the output function of the latter can be thought of as a special case. The reference to θ as a threshold then becomes more plausible as it takes on the role of the same parameter in the TLU.

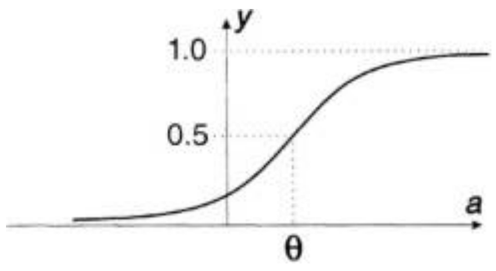


Figure 2.6 Example of squashing function—the sigmoid.

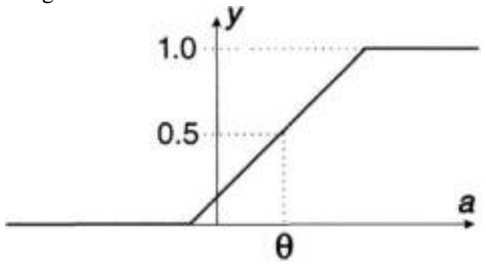


Figure 2.7 Piecewise-linear approximation of sigmoid.

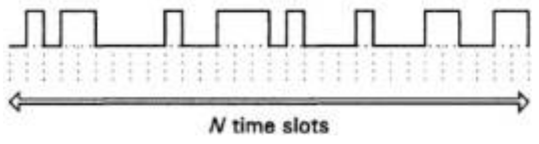


Figure 2.8 Stream of output pulses from a stochastic node.

Artificial neurons or units that use the sigmoidal output relation are referred to as being of the *semilinear* type. The activation is still given by Equation (2.2) but now the output is given by (2.4). They form the bedrock of much work in neural nets since the smooth output function facilitates their mathematical description. The term "semilinear" comes from the fact that we may approximate the sigmoid by a continuous, piecewise-linear function, as shown in Figure 2.7. Over a significant region of interest, at intermediate values of the activation, the output function is a linear relation with non-zero slope.

As an alternative to using continuous or *analogue* signal values, we may emulate the real neuron and encode a signal as the frequency of the occurrence of a "1" in a pulse stream as shown in Figure 2.8.

Time is divided into discrete "slots" and each slot is filled with either a 0 (no pulse) or a 1 (pulse). The unit output is formed in exactly the same way as before but, instead of sending the value of the sigmoid function directly, we interpret it as the probability of emitting a pulse or "1". Processes that are governed by probabilistic laws are referred to as *stochastic* so that these nodes might be dubbed *stochastic semilinear* units, and they produce signals quite close in general appearance to those of real neurons. How are units downstream that receive these signals supposed to interpret their inputs? They must now integrate over some number, *N,* of time slots. Thus, suppose that the afferent node is generating pulses with probability y. The expected value of the number of pulses over this time is yN but, in general, the number actually produced, N_1, will not necessarily be equal to this. The best estimate a node receiving these signals can make is the fraction, N_1/N, of 1s during its integration time. The situation is like that in a coin tossing experiment. The underlying probability of obtaining a "head" is 0. 5, but in any particular sequence of tosses the number of heads N_h is not necessarily one-half of the total. As the number N of tosses increases, however, the fraction N_h/N will eventually approach 0.5.

2.5 Introducing time

Although time reared its head in the last section, it appeared by the back door, as it were, and was not intrinsic to the dynamics of the unit—we could choose *not* to integrate, or, equivalently, set N=1. The way to model the temporal summation of PSPs at the axon hillock is to use the rate of change of the activation as the fundamental defining quantity, rather than the activation itself. A full treatment requires the use of a branch of mathematics known as the *calculus* but the resulting behaviour may be described in a reasonably straightforward way. We shall, however, adopt the calculus notation *dx/dt,* for the rate of change of a quantity x. It cannot be overemphasized that this is to be read as a single symbolic entity, *"dx/dt",* and not

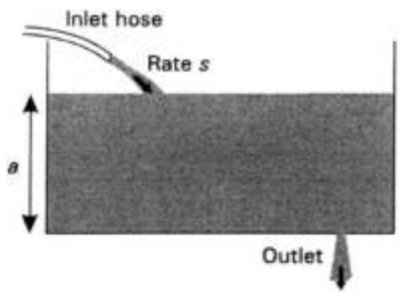


Figure 2.9 Water tank analogy for leaky integrators.

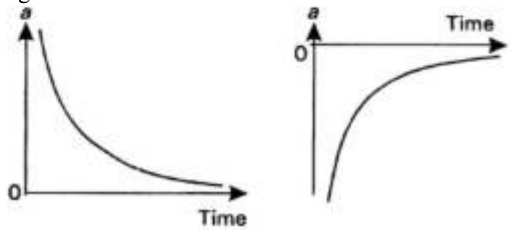


Figure 2.10 Activation decay in leaky integrator.

as *dx* divided by *dt*. To avoid confusion with the previous notation it is necessary to introduce another symbol for the weighted sum of inputs, so we define

$$s = \sum_{i=1}^{n} w_i x_i \tag{2.5}$$

The rate of change of the activation, *da/dt,* is then defined by

$$\frac{da}{dt} = -\alpha a + \beta s \tag{2.6}$$

where α (alpha) and β (beta) are positive constants. The first term gives rise to activation *decay,* while the second represents the input from the other units. As usual the output y is given by the sigmoid of the activation, $y=\sigma(a)$. A unit like this is sometimes known as a *leaky integrator* for reasons that will become apparent shortly.

There is an exact physical analogue for the leaky integrator with which we are all familiar. Consider a tank of water that has a narrow outlet near the base and that is also being fed by hose or tap as shown in Figure 2.9 (we might think of a bathtub, with a smaller drainage hole than is usual). Let the rate at which the water is flowing through the hose be s litres per minute and let the depth of water be a. If the outlet were plugged, the rate of change of water level would be proportional to s, or $da/dt=\beta s$ where β is a constant. Now suppose there is no inflow, but the outlet is working. The rate at which water leaves is directly proportional to the water pressure at the outlet, which is, in turn, proportional to the depth of water α in the tank. Thus, the rate of water emission may be written as αa litres per minute where α is some constant. The water level is now decreasing so that its rate of change is now negative and we have $da/dt=-\alpha a$. If both hose and outlet are functioning then da/dt is the sum of contributions from both, and its governing equation is just the same as that for the neural activation in (2.6). During the subsequent discussion it might be worth while referring back to this analogy if the reader has any doubts about what is taking place.

Returning to the neural model, the activation can be negative or positive (whereas the water level is always positive in the tank). Thus, on putting $s=0$, so that the unit has no external input, there are two cases:

(a) $a>0$. Then $da/dt<0$. That is, the rate of change is negative, signifying a decrease of a with time.
(b) $a<0$. Then $da/dt>0$. That is, the rate of change is positive, signifying an increase of a with time.

These are illustrated in Figure 2.10, in which the left and right sides correspond to cases (a) and (b) respectively In both instances the activity gradually approaches its resting value of zero. It is this decay process that leads to the “leaky” part of the unit’s name. In a TLU or semilinear node, if we withdraw input, the activity immediately becomes zero. In the new model, however, the unit has a kind of short-term memory of its previous input before it was withdrawn. Thus, if this was negative, the activation remains negative for a while afterwards, with a corresponding condition holding for recently withdrawn positive input.

Suppose now that we start with activation zero and no input, and supply a constant input $s=1$ for a time t before withdrawing it again. The activation resulting from this is shown in Figure 2.11. The activation starts to increase but does so rather sluggishly. After s is taken down to zero, a decays in the way described above. If s had been maintained long enough,

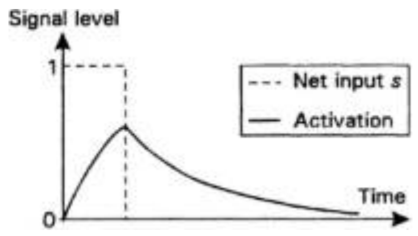


Figure 2.11 Input pulse to leaky integrator.

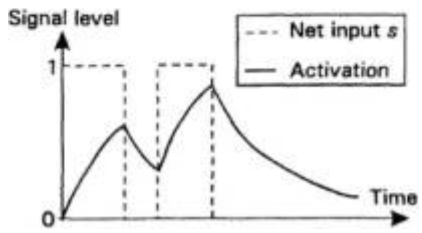


Figure 2.12 Leaky-integrator activation (solid line) for two square input pulses (dashed line).

then a would have eventually reached a constant value. To see what this is we put da/dt=0, since this is a statement of there being no rate of change of a, and a is constant at some equilibrium value a_{eqm}. Putting da/dt=0 in (2.6) gives

$$a_{eqm} = \left(\frac{\beta}{\alpha}\right) s \qquad (2.7)$$

that is, a constant fraction of s. If $\alpha=\beta$ then $a_{eqm}=s$. The speed at which a can respond to an input change may be characterized by the time taken to reach some fraction of a_{eqm} ($0.75a_{eqm}$, say) and is called the *rise-time*.

Suppose now that a further input pulse is presented soon after the first has been withdrawn. The new behaviour is shown in Figure 2.12. Now the activation starts to pick up again as the second input signal is delivered and, since a has not had time to decay to its resting value in the interim, the peak value obtained this time is larger than before. Thus the two signals interact with each other and there is temporal summation or integration (the "integrator" part of the unit's name). In a TLU, the activation would, of course, just be equal to s. The value of the constants α and β govern the decay rate and rise-time respectively and, as they are increased, the decay rate increases and the rise-time falls. Keeping $\alpha=\beta$ and letting both become very large therefore allows a to rise and fall very quickly and to reach equilibrium at s. As these constants are increased further, the resulting behaviour of a becomes indistinguishable from that of a TLU, which can therefore be thought of as a special case of the leaky integrator with very large constants α, β (and, of course, very steep sigmoid).

Leaky integrators find their main application in self-organizing nets (Ch. 8). They have been studied extensively by Stephen Grossberg who provides a review in Grossberg (1988). What Grossberg calls the "additive STM model" is essentially the same as that developed here, but he also goes on to describe another—the "shunting STM" neuron—which is rather different.

This completes our first foray into the realm of artificial neurons. It is adequate for most of the material in the rest of this book but, to round out the story, Chapter 10 introduces some alternative structures.

2.6 Summary

The function of real neurons is extremely complex. However, the essential information processing attributes may be summarized as follows. A neuron receives input signals from many other (afferent) neurons. Each such signal is modulated (by the synaptic mechanism) from the voltage spike of an action potential into a continuously variable (graded) postsynaptic potential (PSP). PSPs are integrated by the dendritic arbors over both space (many synaptic inputs) and time (PSPs do not decay to zero instantaneously). PSPs may be excitatory or inhibitory and their integrated result is a change in the membrane potential at the axon hillock, which may serve to depolarize (excite or activate) or hyperpolarize (inhibit) the neuron. The dynamics of the membrane under these changes are complex but may be described in many instances by supposing that there is a membrane-potential threshold, beyond which an action potential is generated and below which no such event takes place. The train of action potentials constitutes the neural "output". They travel away from the cell body along the axon until they reach axon terminals (at synapses) upon which the cycle of events is initiated once again. Information is encoded in many ways in neurons but a common method is to make use of the frequency or rate of production of action potentials.

The integration of signals over space may be modelled using a linear weighted sum of inputs. Synaptic action is then supposed to be equivalent to multiplication by a weight. The TLU models the action potential by a simple threshold mechanism that allows two signal levels (0 or 1). The rate of firing may be represented directly in a semilinear node by allowing a continuous-valued output or (in the stochastic variant) by using this value as a probability for the production of signal pulses. Integration over time is catered for in the leaky-integrator model. All artificial neurons show robust behaviour under degradation of input signals and hardware failure.

2.7 Notes

1. The millivolt (mV) is one-thousandth of a volt.
2. After George Boole who developed a formal logic with two values denoting “True” and “False”.
3. Scientific calculators should have this as one of their special purpose buttons.

Chapter Three
TLUs, linear separability and vectors

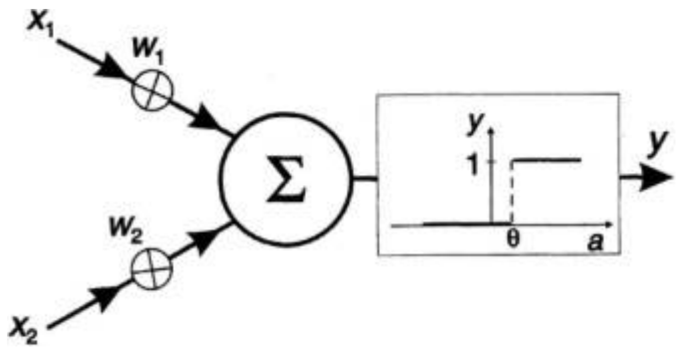


Figure 3.1 Two-input TLU.

The simplest artificial neuron presented in the last chapter was the threshold logic unit or TLU. In this chapter we shall discuss a geometric context for describing the functionality of TLUs and their networks that has quite general relevance for the study of all neural networks. In the process it will be necessary to introduce some mathematical concepts about vectors. These are also of general importance and so their properties are described in some detail. Readers already familiar with this material may still wish to skim Section 3.2 to become acquainted with our notation.

3.1 Geometric interpretation of TLU action

In summary, a TLU separates its input patterns into two categories according to its binary response ("0" or "1") to each pattern. These categories may be thought of as regions in a multidimensional space that are separated by the higher dimensional equivalent of a straight line or plane.

These ideas are now introduced step by step and in a way that should help put to rest any concerns about "higher dimensionality" and "multidimensional spaces".

3.1.1 Pattern classification and input space

Consider a two-input TLU with weights w_1=1, w_2=1 and threshold 1.5, as shown in Figure 3.1. The responses to the four possible Boolean inputs are shown in Table 3.1. The TLU may be thought of as *classifying* its input patterns into two groups: those that give output "1" and those that give output "0". Each input pattern has two *components,* x_1, x_2. We may therefore represent these patterns in a two-dimensional space as shown in Figure 3.2.

Each pattern determines a point in this so-called *pattern space* by using its

Table 3.1 Functionality of two-input TLU example.

x_1	x_2	Activation	Output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

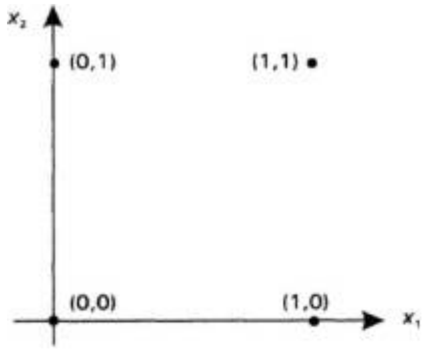


Figure 3.2 Two-input patterns in pattern space.

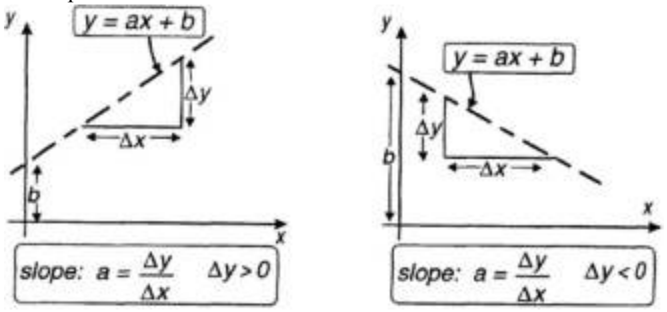


Figure 3.3 Straight line graphs.

component values as space co-ordinates—just as grid references can locate points in physical space on a normal geographical map. In general, for *n* inputs, the pattern space will be *n* dimensional. Clearly, for $n>3$ the pattern space cannot be drawn or represented in physical space. This is not a problem. The key is that all relationships between patterns can be expressed either geometrically, as in Figure 3.2, or algebraically using the notion of *vectors*. We can then gain insight into pattern relationships in two dimensions (2D), reformulate this in vector form and then simply carry over the results to higher dimensions. This process will become clearer after it has been put to use later. All the necessary tools for using vectors are introduced in this chapter; their appreciation will significantly increase any understanding of neural nets.

We now develop further the geometric representation of our two-input TLU.

3.1.2 The linear separation of classes

Since the critical condition for classification occurs when the activation equals the threshold, we will examine the geometric implication of this. For two inputs, equating θ and *a* gives

$$w_1x_1 + w_2x_2 = \theta \tag{3.1}$$

Subtracting w_1x_1 from both sides

$$w_2x_2 = -w_1x_1 + \theta \tag{3.2}$$

and dividing both sides by w_2 gives

$$x_2 = -\left(\frac{w_1}{w_2}\right)x_1 + \left(\frac{\theta}{w_2}\right) \tag{3.3}$$

This is of the general form

$$x_2 = ax_1 + b \tag{3.4}$$

where *a* and *b* are constants. This equation describes a straight line with *slope a* and *intercept b* on the x_2 axis. This is illustrated in Figure 3.3 where the graph of the equation $y=ax+b$ has been plotted for two sets of values of *a, b*. In each case the slope is given by the change Δy that occurs in *y* when a positive change Δx is made in *x* ("Δ" is Greek upper case delta and usually signifies a change in a quantity). As an example, in motoring, to describe a hill as having a "one-in-ten" slope implies you have to travel 10 metres to go up 1 metre and the hill therefore has a slope of magnitude 1/10. In the notation introduced here, there is a Δx of 10 associated with a Δy of 1. In the left hand part of the figure, *y* is increasing with *x*. Therefore, $\Delta y>0$ when a positive change Δx is made, and so the slope *a* is positive. The right hand part of the figure shows *y* decreasing with *x* so that $\Delta y<0$ when *x* is increased, resulting in a negative slope.

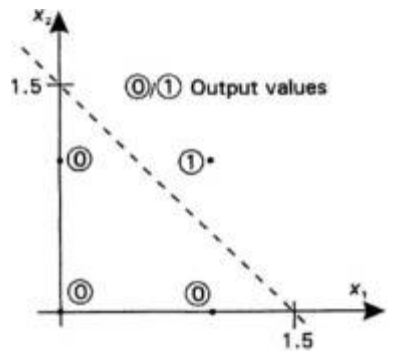


Figure 3.4 Decision line in two-input example.

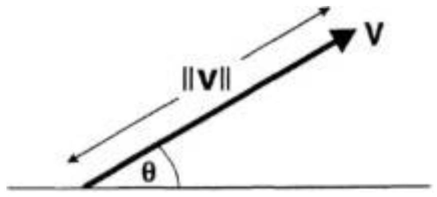


Figure 3.5 A vector.

For the TLU example, inserting the values of w_1, w_2, θ in (3.3) we obtain $a=-1$, $b=1.5$ as shown in Figure 3.4, which also shows the output of the TLU for each pattern. The two classes of TLU output are separated by the line produced in this way so that the 1s (there is only one of them) and 0s lie on opposite sides of the line; we therefore talk of this as the *decision line*. Clearly, it is always possible to partition the two classes in 2D by drawing some kind of line—the point here is that the line is a straight one having no kinks or bends. It turns out that this is not just a fortuitous result made possible by our choice of weights and threshold. It holds true for any two-input TLU. This distinction is clearer in 3D where, quite generally, we can define a *decision surface* that may have to be highly convoluted but a TLU will necessarily be associated with a flat *decision plane*.

Further, it is possible to generalize this result (in its algebraic form) to TLUs with an arbitrary number, n say, of inputs; that is, it is always possible to separate the two output classes of a TLU by the n-dimensional equivalent of a straight line in 2D or, in 3D, a plane. In n dimensions this is referred to as the *decision hyperplane*. (The "hyper-" is sometimes dropped even when $n>3$). Because TLUs are intimately related to linear relations like (3.3) (and their generalization) we say that TLUs are *linear classifiers* and that their patterns are *linearly separable*. The converse of our result is also true: any binary classification that cannot be realized by a linear decision surface cannot be realized by a TLU.

We now try to demonstrate these results using the machinery of vectors. These ideas will also have general applicability in our discussion of nets throughout.

3.2 Vectors

Vectors are usually introduced as representations of quantities that have magnitude and direction. For example, the velocity of the wind is defined by its speed and direction. On paper we may draw an arrow whose direction is the same as that of the wind and whose length is proportional to its speed. Such a representation is the basis for some of the displays on televised weather reports, and we can immediately see when there will be high winds, as these are associated with large arrows. A single vector is illustrated in Figure 3.5, which illustrates some notation.

Vectors are usually denoted in printed text by bold face letters (e.g. **v**), but in writing them by hand we can't use bold characters and so make use of an underline as in $\underline{v}$. The magnitude (or length) of v will be denoted by $\|\mathbf{v}\|$ but is also sometimes denoted by the same letter in italic face (e.g. v). In accordance with our geometric ideas a vector is now defined by the pair of numbers ($\|\mathbf{v}\|$, θ) where θ is the angle the vector makes with some reference direction. Vectors are to be distinguished from simple numbers or *scalars,* which have a value but no direction.

In order to generalize to higher dimensions, and to relate vectors to the ideas of pattern space, it is more convenient to describe vectors with respect to a rectangular or *cartesian* co-ordinate system like the one used for the TLU example in 2D. That is, we give the projected lengths of the vector onto two perpendicular axes as shown in Figure 3.6.

The vector is now described by the pair of numbers v_1, v_2. These numbers are its *components* in the chosen co-ordinate system. Since they completely determine the vector we may think of the vector itself as a pair of component values and write $\mathbf{v}=(v_1, v_2)$. The vector is now an ordered list of numbers. Note that the ordering is important, since (1, 3) is in a different

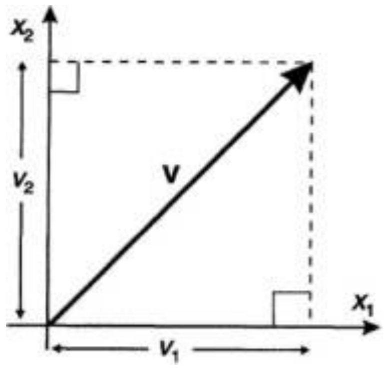


Figure 3.6 Vector components.

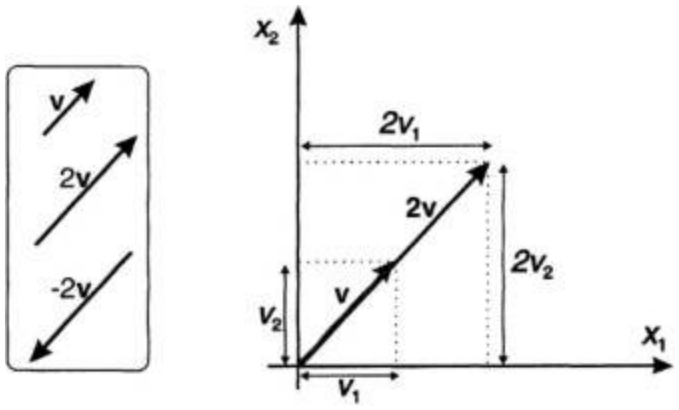


Figure 3.7 Scalar multiplication of a vector.

direction from (3, 1). It is this algebraic definition that immediately generalizes to more than 2D. An n-dimensional vector is simply an ordered list of *n* numbers, **v**=$(v_1, v_2,..., v_n)$. They become of interest when rules are defined for combining them and multiplying them by numbers or scalars (see below). To motivate the following technical material, we note that there are two vectors of immediate concern to us—the *weight vector* $(w_1, w_2,..., w_n)$ and the *input vector* $(x_1, x_2,..., x_n)$ for artificial neurons.

3.2.1
Vector addition and scalar multiplication

Multiplying a vector by a number (scalar) *k* simply changes the length of the vector by this factor so that if *k*=2, say, then we obtain a vector of twice the length. Multiplying by a negative number results in a reversal of vector direction and a change in length required by the number's magnitude—see Figure 3.7. In component terms, if a vector in 2D, **v**=(v_1, v_2), is multiplied by *k*, then the result[1] v′ has components (kv_1, kv_2). This can be seen in the right hand side of Figure 3.7 where the original vector v is shown stippled. Generalizing to *n* dimensions we *define* vector multiplication by *k***v**=$(kv_1, kv_2,..., kv_n)$.

Geometrically, two vectors may be added in 2D by simply appending one to the end of the other as shown in Figure 3.8. Notice that a vector may be drawn anywhere in the space as long as its magnitude and direction are preserved. In terms of the components, if **w**=**u**+**v**, then $w_1=u_1+v_1$, $w_2=u_2+v_2$. This lends itself to generalization in *n* dimensions in a straightforward way. Thus, if u, v are now vectors in *n* dimensions with sum **w**, **w**=$(u_1+v_1, u_2+v_2,..., u_n+v_n)$. Note that **u**+**v**=**v**+**u.**

Vector subtraction is defined via a combination of addition and scalar multiplication so that we interpret **u**−**v** as **u**+(−1)**v**, giving the addition of u and a reversed copy of v (see Fig. 3.9). The left hand side of the figure shows the original vectors **u** and **v**. The construction for subtraction is shown in the centre and the right hand side shows how, by making use of the symmetry of the situation, the resulting vector **w** may be drawn as straddling **u** and **v** themselves.

3.2.2
The length of a vector

For our prototype in 2D, the length of a vector is just its geometrical length in the plane. In terms of its components, this is given by applying Pythagoras's theorem to the triangle shown in Figure 3.10, so that

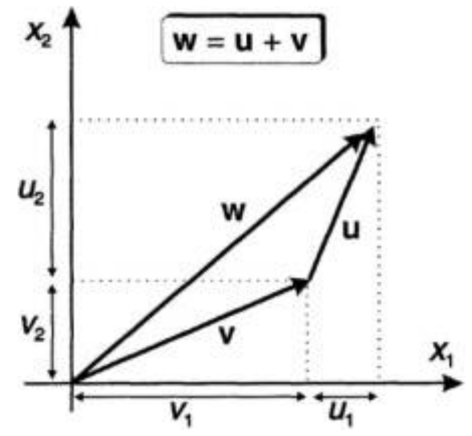


Figure 3.8 Vector addition.

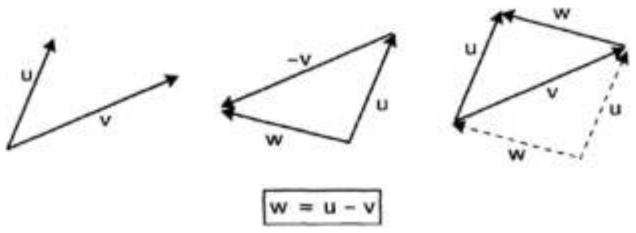


Figure 3.9 Vector subtraction.

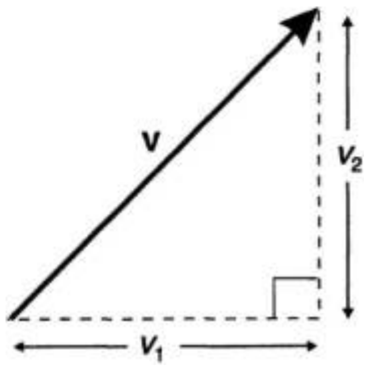


Figure 3.10 Obtaining the length of a vector.

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2} \tag{3.5}$$

In *n* dimensions, the length is defined by the natural extension of this, so that

$$\|\mathbf{v}\| = \left[\sum_{i=1}^{n} v_i^2\right]^{\frac{1}{2}} \tag{3.6}$$

where the exponent of $\frac{1}{2}$ outside the square brackets is a convenient way of denoting the operation of square root.

3.2.3 Comparing vectors—the inner product

In several situations in our study of networks it will be useful to have some measure of how well aligned two vectors are—that is, to know whether they point in the same or opposite directions. The vector *inner product* allows us to do just this. This section relies on the trigonometric function known as the *cosine* and so, for those readers who may not be familiar with this, it is described in an appendix.

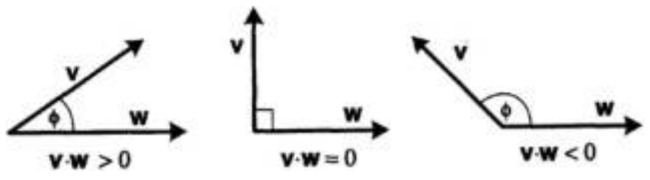
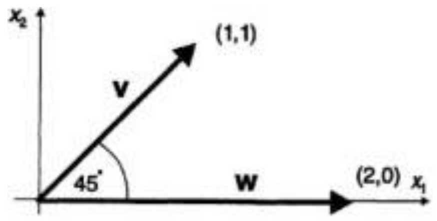


Figure 3.11 Inner product examples.



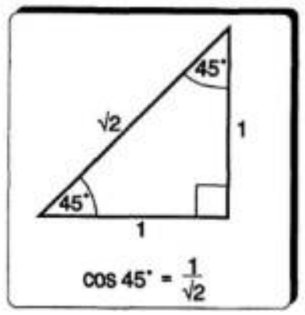


Figure 3.12 Vectors at 45°.

Inner product—geometric form

Suppose two vectors **v** and **w** are separated by an angle *ø*. Define the *inner product* **v·w** of the two vectors by the product of their lengths and the cosine of *ø;* that is,

$$\mathbf{v}\cdot\mathbf{w} = \|\mathbf{v}\|\|\mathbf{w}\|\cos\phi \tag{3.7}$$

This is pronounced "v dot w" and is also known as the scalar product since its result is a number (rather than another vector). Note that **v·w=w·v**.

What is the significance of this definition? Essentially (as promised) it tells us something about the way two vectors are aligned with each other, which follows from the properties of the cosine function. To see this, fix w but allow v to vary its direction (but not its lengths) as shown in Figure 3.11. Then, if the lengths are fixed, **v·w** can only depend on cos*ø*. When 0<*ø*<90°, the cosine is positive and so too, therefore, is the inner product. However, as the angle approaches 90°, the cosine diminishes and eventually reaches zero. The inner product follows in sympathy with this and, when the two vectors are at right angles they are said to be *orthogonal* with **v·w=0**. Thus, if the vectors are well aligned or point in roughly the same direction, the inner product is close to its largest positive value of ||**v**|| ||**w**||. As they move apart (in the angular sense) their inner product decreases until it is zero when they are orthogonal. As *ø* becomes greater than 90°, the cosine becomes progressively more negative until it reaches −1. Thus, ||**v**|| ||**w**|| also behaves in this way until, when *ø*=180°, it takes on its largest negative value of −||**v**|| ||**w**||. Thus, if the vectors are pointing in roughly opposite directions, they will have a relatively large *negative* inner product.

Note that we only need to think of angles in the range 0<*ø*<180° because a value of *ø* between 180° and 360° is equivalent to an angle given by 360−*ø*.

Inner product—algebraic form

Consider the vectors **v**=(1, 1) and **w**=(0, 2) shown in Figure 3.12 where the angle between them is 45°. An inset in the figure shows a right-angled triangle with its other angles equal to 45°. The hypotenuse, *h,* has been calculated from Pythagoras's theorem to be $\sqrt{1^2+1^2} = \sqrt{2}$ and, from the definition of the cosine (A.1), it can then be seen that $\cos 45° = 1/\sqrt{2}$. To find the inner product of the two vectors in Figure 3.12, we note that $\|\mathbf{v}\| = \sqrt{2}, \|\mathbf{w}\| = 2,$ so that $\mathbf{v}\cdot\mathbf{w} = \sqrt{2}\times 2\times 1/\sqrt{2} = 2$. We now introduce an equivalent, algebraic definition of the inner product that lends itself to generalization in *n* dimensions.

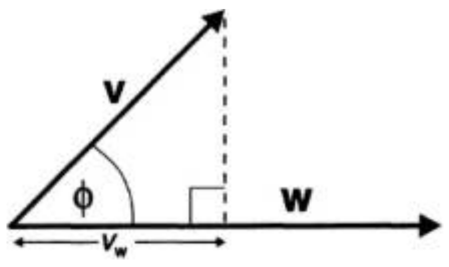


Figure 3.13 Vector projections.

Consider the quantity **v** ○ **w** defined in 2D by

$$\mathbf{v} \circ \mathbf{w} = v_1 w_1 + v_2 w_2 \tag{3.8}$$

The form on the right hand side should be familiar—substituting x for v we have the activation of a two-input TLU. In the example above, substituting the component values gives **v** ○ **w=2** which is the same as v·w. The equivalence of what we have called **v** ○ **w** and the geometrically defined inner product is not a chance occurrence resulting from the particular choice of numbers in our example. It is a general result (Rumelhart et al. 1986b) (which will not be proved here) and it means that we may write $\mathbf{v}\cdot\mathbf{w}=v_1w_1+v_2w_2$ for any vectors **v, w** in 2D. The form in (3.8) immediately lends itself to generalization in n dimensions so that we define the dot product of two n-dimensional vectors **v, w** as

$$\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{n} w_i v_i \tag{3.9}$$

We shall interpret the value obtained in this way just as we did in 2D. Thus, if it is positive then the two vectors are, in some sense, roughly "lined up" with each other, if it is negative then they are "pointing away from" each other and, if it is zero, the vectors are at "right angles". No attempt should be made to visualize this in n dimensions; rather, think of its analogue in 2D as a schematic or cartoon representation of what is happening. The situation is a little like using pictures in 2D to represent scenes in 3D—the picture is not identical to the objects it depicts in 3D, but it may help us think about their geometrical properties.

Finally, what happens if **v=w**? Then we have

$$\mathbf{v} \cdot \mathbf{v} = \sum_{i} v_i v_i = \|\mathbf{v}\|^2 \tag{3.10}$$

so that the square length of vector is the same as the inner product of the vector with itself.

Vector projection

There is one final concept that we will find useful. Consider the two vectors **v, w** in Figure 3.13 and suppose we ask the question—how much of **v** lies in the direction of **w**? Formally, if we drop a perpendicular from **v** onto **w** what is the length of the line segment along w produced in this way? This segment is called the *projection* v_w of **v** *onto* **w** and, using the definition of cosine, we have $v_w=\|\mathbf{v}\|\cos ø$. We can reformulate this, using the inner product, in a way suitable for generalization. Thus, we write

$$\begin{aligned} v_w &= \frac{\|\mathbf{v}\|\|\mathbf{w}\|\cos\phi}{\|\mathbf{w}\|} \\ &= \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{w}\|} \end{aligned} \tag{3.11}$$

3.3 TLUs and linear separability revisited

Our discussion of vectors was motivated by the desire to prove that the connection between TLUs and linear separability is a universal one, independent of the dimensionality of the pattern space. We are now in a position to show this, drawing on the ideas developed in the previous section. Using the definition of the inner product (3.9) the activation a of an n-input TLU may now be expressed as

$$a = \mathbf{w} \cdot \mathbf{x} \tag{3.12}$$

The vector equivalent to (3.1) now becomes

$$\mathbf{w} \cdot \mathbf{x} = \theta \tag{3.13}$$

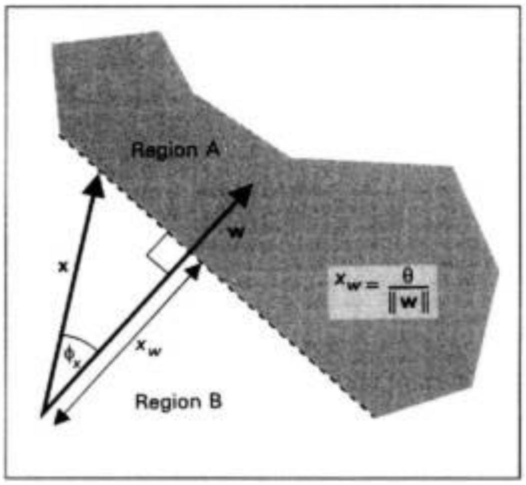


Figure 3.14 Projection of x as decision line.

As in the example in 2D, we expect deviations either side of those x that satisfy this relation to result in different output for the TLU. We now formalize what is meant by "either side" a little more carefully. Our strategy is to examine the case in 2D geometrically to gain insight and then, by describing it algebraically, to generalize to *n* dimensions.

In general, for an arbitrary **x,** the projection of **x** onto **w** is given by

$$x_w = \frac{\mathbf{w}\cdot\mathbf{x}}{\|\mathbf{w}\|} \tag{3.14}$$

If, however, we impose the constraint implied by (3.13), we have

$$x_w = \frac{\theta}{\|\mathbf{w}\|} \tag{3.15}$$

So, assuming **w** and *θ* are constant, the projection x_w is constant and, in 2D, **x** must actually lie along the perpendicular to the weight vector, shown as a dashed line in Figure 3.14. Therefore, in 2D, the relation $\mathbf{w}\cdot\mathbf{x}=\theta$ defines a straight line. However, since we have used algebraic expressions that are valid in *n* dimensions throughout, we can generalize and use this to *define* the *n*-dimensional equivalent of a straight line—a hyperplane—which is perpendicular to the weight vector **w**. When *x* lies on the hyperplane, $\mathbf{w}\cdot\mathbf{x}=\theta$, and the TLU output rule states that y=1; it remains to see what happens on each side of the line.

Suppose first that $x_w>\theta/\|\mathbf{w}\|$; then the projection is longer than that in Figure 3.14 and **x** must lie in region A (shown by the shading). Comparison of (3.14) and (3.15) shows that, in this case, $\mathbf{w}\cdot\mathbf{x}>\theta$, and so y=1. Conversely, if $x_w<\theta/\|\mathbf{w}\|$, the projection is shorter than that in Figure 3.14 and **x** must lie in region B. The implication is now that $\mathbf{w}\cdot\mathbf{x}<\theta$, and so y=0. The diagram can only show part of each region and it should be understood that they are, in fact, of infinite extent so that any point in the pattern space is either in A or B. Again these results are quite general and are independent of the number *n* of TLU inputs.

To summarize: we have proved two things:

(a) The relation $\mathbf{w}\cdot\mathbf{x}=\theta$ defines a hyperplane (*n*-dimensional "straight line") in pattern space which is perpendicular to the weight vector. That is, any vector wholly *within* this plane is orthogonal to **w**.
(b) On one side of this hyperplane are all the patterns that get classified by the TLU as a "1", while those that get classified as a "0" lie on the other side of the hyperplane.

To recap on some points originally made in Section 3.1.2, the hyperplane is the decision surface for the TLU. Since this surface is the *n*-dimensional version of a straight line the TLU is a *linear* classifier. If patterns cannot be separated by a hyperplane then they cannot be classified with a TLU.

One assumption has been made throughout the above that should now be made explicit. Thus, Figure 3.14 shows a positive projection x_w, which implies a positive threshold. For a negative threshold *θ*, the projection constraint (3.15) now implies that $x_w<0$, since $\|\mathbf{w}\|$ is always positive. Therefore $\mathbf{w}\cdot\mathbf{x}<0$ for those **x** that lie on the decision line and they must point away from **w** as shown in the left half of Figure 3.15. A typical instance of the use of a negative threshold is shown in the right hand part of the figure. Notice that the weight vector always points towards the region of 1s, which is consistent with the TLU rule: $\mathbf{w}\cdot\mathbf{x}>0$ implies y=1.

3.4 Summary

The function of a TLU may be represented geometrically in a pattern space. In this space, the TLU separates its inputs into two classes (depending on the output "1" or "0"), which may be separated by a hyperplane (the *n*-dimensional equivalent of a

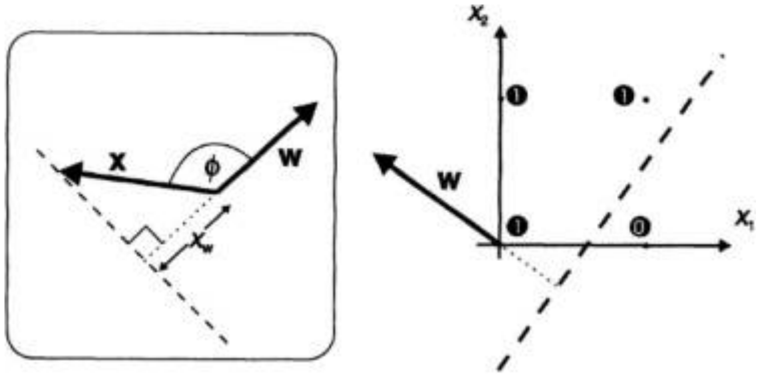


Figure 3.15 Projection with negative threshold.

straight line in 1D or a plane in 2D). The formal description of neuron behaviour in pattern space is greatly facilitated by the use of vectors, which may be thought of as the generalization of directed arrows in 2D or 3D. A key concept is that of the dot product **w·x** of two vectors **w** and **x**. If the lengths of the two vectors are held fixed, then the dot product tells us something about the "angle" between the vectors. Vector pairs that are roughly aligned with each other have a positive inner product, if they point away from each other the inner product is negative, and if they are at right angles (orthogonal) it is zero. The significance of all this is that the activation of a TLU is given by the dot product of the weight and input vectors, a=**w·x,** so that it makes sense to talk about a neuron computing their relative alignment. Our first application of this was to prove the linear separability of TLU classes. However, the geometric view (and the dot product interpretation of activation) will, quite generally, prove invaluable in gaining insight into network function.

3.5
Notes

1. The small dash symbol is pronounced "prime" so one reads v′ as "v-prime".

Chapter Four
Training TLUs: the perceptron rule

4.1
Training networks

This chapter introduces the concept of training a network to perform a given task. Some of the ideas discussed here will have general applicability, but most of the time refer to the specifics of TLUs and a particular method for training them. In order for a TLU to perform a given classification it must have the desired decision surface. Since this is determined by the weight vector and threshold, it is necessary to adjust these to bring about the required functionality. In general terms, adjusting the weights and thresholds in a network is usually done via an iterative process of repeated presentation of examples of the required task. At each presentation, small changes are made to weights and thresholds to bring them more in line with their desired values. This process is known as *training* the net, and the set of examples as the *training set*. From the network's viewpoint it undergoes a process of learning, or adapting to, the training set, and the prescription for how to change the weights at each step is the *learning rule*. In one type of training (alluded to in Ch. 1) the net is presented with a set of input patterns or vectors $\{\mathbf{x}_i\}$ and, for each one, a corresponding desired output vector or target $\{\mathbf{t}_i\}$. Thus, the net is supposed to respond with $\mathbf{t}_k$, given input $\mathbf{x}_k$ for every k. This process is referred to as *supervised* training (or learning) because the network is told or supervised at each step as to what it is expected to do.

We will focus our attention in this chapter on training TLUs and a related node, the *perceptron,* using supervised learning. We will consider a single node in isolation at first so that the training set consists of a set of pairs $\{\mathbf{v}, t\}$, where $\mathbf{v}$ is an input vector and t is the target class or output ("1" or "0") that $\mathbf{v}$ belongs to.

4.2
Training the threshold as a weight

In order to place the adaptation of the threshold on the same footing as the weights, there is a mathematical trick we can play to make it look like a weight. Thus, we normally write $\mathbf{w}\cdot\mathbf{x}\geq\theta$ as the condition for output of a "1". Subtracting θ from both sides gives $\mathbf{w}\cdot\mathbf{x}-\theta\geq 0$ and making the minus sign explicit results in the form $\mathbf{w}\cdot\mathbf{x}+(-1)\theta\geq 0$. Therefore, we may think of the threshold as an extra weight that is driven by an input constantly tied to the value −1. This leads to the negative of the threshold being referred to sometimes as the *bias*. The weight vector, which was initially of dimension n for an n-input unit, now becomes the $(n+1)$-dimensional vector $w_1, w_2,\ldots, w_n, \theta$. *We* shall call this the *augmented* weight vector, in contexts where confusion might arise, although this terminology is by no means standard. Then for all TLUs we may express the node function as follows[1]:

$$\begin{array}{lll} \mathbf{w}\cdot\mathbf{x}\geq 0 & \Rightarrow & y=1 \\ \mathbf{w}\cdot\mathbf{x}<0 & \Rightarrow & y=0 \end{array} \tag{4.1}$$

Putting $\mathbf{w}\cdot\mathbf{x}=0$ now defines the decision hyperplane, which, according to the discussion in Chapter 3, is orthogonal to the (augmented) weight vector. The zero-threshold condition in the augmented space means that the hyperplane passes through the origin, since this is the only way that allows $\mathbf{w}\cdot\mathbf{x}=0$. We now illustrate how this modification of pattern space works with an example in 2D, but it is quite possible to skip straight to Section 4.3 without any loss of continuity.

Consider the two-input TLU that outputs a "1" with input (1, 1) and a "0" for all other inputs so that a suitable (non-augmented) weight vector is (1/2, 1/2) with threshold 3/4. This is shown in Figure 4.1 where the decision line and weight vector have been shown quantitatively. That the decision line goes through the points $\mathbf{x}_1$=(1/2, 1) and $\mathbf{x}_2$=(1, 1/2) may be easily verified since according to (3.8) $\mathbf{w}\cdot\mathbf{x}_1=\mathbf{w}\cdot\mathbf{x}_2=3/4=\theta$. For the augmented pattern space we have to go to 3D as shown in Figure 4.2. The previous two components x_1, x_2 are now drawn in the horizontal plane while a third component x_3 has been introduced, which is shown as the vertical axis. All the patterns to the TLU now have the form $(x_1, x_2, -1)$ since the third input is tied to the constant value of −1. The augmented weight vector now has a third component equal to the threshold and

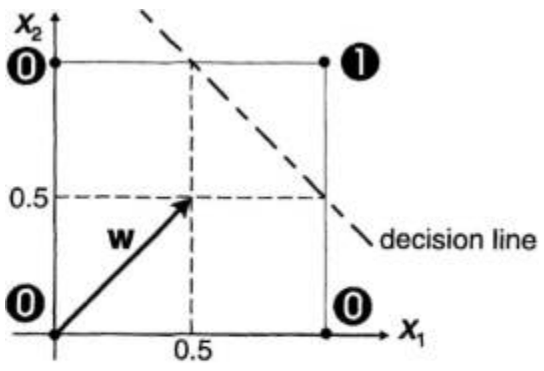


Figure 4.1 Two-dimensional TLU example.

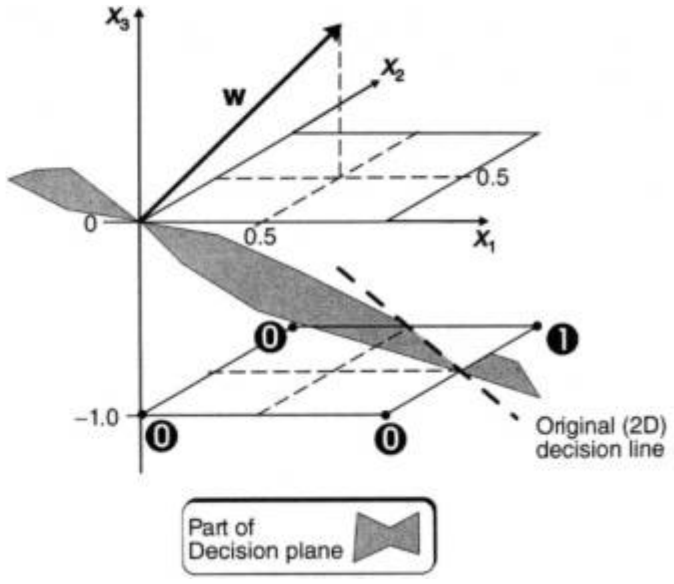


Figure 4.2 Two-dimensional example in augmented pattern space.

is perpendicular to a decision plane that passes through the origin. The old decision line in 2D is formed by the intersection of the decision plane and the plane containing the patterns.

4.3 Adjusting the weight vector

We now suppose we are to train a single TLU with augmented weight vector **w** using the training set consisting of pairs like **v**, *t*. The TLU may have any number of inputs but we will represent what is happening in pattern space in a schematic way using cartoon diagrams in 2D.

Suppose we present an input vector **v** to the TLU with desired response or target *t*=1 and, with the current weight vector, it produces an output of *y*=0. The TLU has misclassified and we must make some adjustment to the weights. To produce a "0" the activation must have been negative when it should have been positive—see (4.1). Thus, the dot product **w·v** was negative and the two vectors were pointing away from each other as shown on the left hand side of Figure 4.3.

In order to correct the situation we need to rotate **w** so that it points more in the direction of **v**. At the same time, we don't want to make too drastic a change as this might upset previous learning. We can achieve both goals by adding a fraction of **v** to **w** to produce a new weight vector **w′**, that is

$$\mathbf{w}' = \mathbf{w} + \alpha \mathbf{v} \tag{4.2}$$

where $0<\alpha<1$, which is shown schematically on the right hand side of Figure 4.3.

Suppose now, instead, that misclassification takes place with the target *t*=0 but *y*=1. This means the activation was positive when it should have been negative as shown on the left in Figure 4.4. We now need to rotate **w** *away* from **v**, which may be effected by *subtracting* a fraction of **v** from **w**, that is

$$\mathbf{w}' = \mathbf{w} - \alpha \mathbf{v} \tag{4.3}$$

as indicated on the left of the figure.

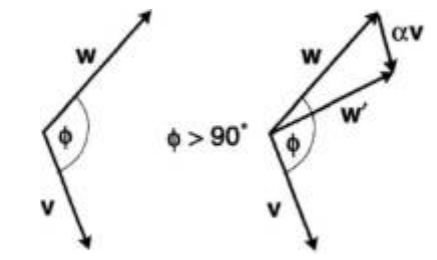


Figure 4.3 TLU misclassification 1–0.

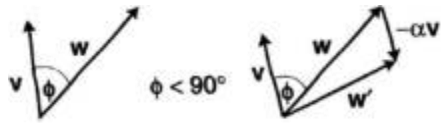


Figure 4.4 TLU misclassification 0–1.

Both (4.2) and (4.3) may be combined as a single rule in the following way:

$$\mathbf{w}' = \mathbf{w} + \alpha(t - y)\mathbf{v} \tag{4.4}$$

This may be written in terms of the change in the weight vector $\Delta\mathbf{w}=\mathbf{w}'-\mathbf{w}$ as follows:

$$\Delta\mathbf{w} = \alpha(t - y)\mathbf{v} \tag{4.5}$$

or in terms of the components

$$\Delta w_i = \alpha(t - y)v_i \quad : \quad i = 1 \text{ to } n+1 \tag{4.6}$$

where $w_{n+1}=\theta$ and $v_{n+1}=-1$ always. The parameter α is called the *learning rate* because it governs how big the changes to the weights are and, hence, how fast the learning takes place. All the forms (4.4, 4.5, 4.6) are equivalent and define the *perceptron training rule*. It is called this rather than the TLU rule because, historically, it was first used with a modification of the TLU known as the perceptron, described in Section 4.4. The learning rule can be incorporated into the overall scheme of iterative training as follows.

```
repeat
    for each training vector pair ($\mathbf{v}$, $t$)
        evaluate the output $y$ when $\mathbf{v}$ is input to the TLU
        if $y \neq t$ then
            form a new weight vector $\mathbf{w}'$ according to (4.4)
        else
            do nothing
        end if
    end for
until $y = t$ for all vectors
```

The procedure in its entirety constitutes the perceptron *learning algorithm*. There is one important assumption here that has not, as yet, been made explicit: the algorithm will generate a valid weight vector for the problem in hand, if one exists. Indeed, it can be shown that this is the case and its statement constitutes the *perceptron convergence theorem:*

> If two classes of vectors *X, Y* are linearly separable, then application of the perceptron training algorithm will eventually result in a weight vector $\mathbf{w}_0$ such that $\mathbf{w}_0$ defines a TLU whose decision hyperplane separates *X* and *Y*.

Since the algorithm specifies that we make no change to **w** if it correctly classifies its input, the convergence theorem also implies that, once $\mathbf{w}_0$ has been found, it remains stable and no further changes are made to the weights. The convergence theorem was first proved by Rosenblatt (1962), while more recent versions may be found in Haykin (1994) and Minsky & Papert (1969).

One final point concerns the uniqueness of the solution. Suppose $\mathbf{w}_0$ is a valid solution to the problem so that $\mathbf{w}_0\cdot\mathbf{x}=0$ defines a solution hyperplane. Multiplying both sides of this by a constant *k* preserves the equality and therefore defines the same hyperplane. We may absorb *k* into the weight vector so that, letting $\mathbf{w}'_0=k\mathbf{w}_0$, we have $\mathbf{w}'_0\cdot\mathbf{x}=k\mathbf{w}_0\cdot\mathbf{x}=0$. Thus, if $\mathbf{w}_0$ is a solution, then so too is $k\mathbf{w}_0$ for any *k* and this entire family of vectors defines the same solution hyperplane.

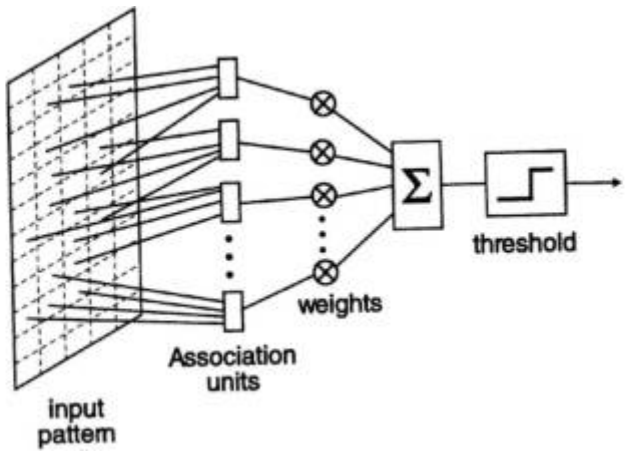


Figure 4.5 The perception.

We now look at an example of the training algorithm in use with a two-input TLU whose initial weights are 0, 0.4, and whose initial threshold is 0.3. It has to learn the function illustrated in Figure 4.1; that is, all inputs produce 0 except for the vector (1, 1). The learning rate is 0.25. Using the above algorithm, it is possible to calculate the sequence of events that takes place on presentation of all four training vectors as shown in Table 4.1.

Table 4.1 Training with the perception rule on a two-input example.

w_1	w_2	θ	x_1	x_2	a	y	t	$\alpha(t-y)$	δw_1	δw_2	$\delta\theta$
0.0	0.4	0.3	0	0	0	0	0	0	0	0	0
0.0	0.4	0.3	0	1	0.4	1	0	−0.25	0	−0.25	0.25
0.0	0.15	0.55	1	0	0	0	0	0	0	0	0
0.0	0.15	0.55	1	1	0.15	0	1	0.25	0.25	0.25	−0.25

Each row shows the quantities required for a single vector presentation. The columns labelled w_1, w_2, θ show the weights and threshold just prior to the application of the vector with components in columns x_1, x_2. The columns marked a and y show the activation and output resulting from input vector (x_1, x_2). The target t appears in the next column and takes part in the calculation of the quantity $\alpha(t-y)$, which is used in the training rule. If this is non-zero then changes are effected in the weights δw_1, δw_2, and threshold $\delta\theta$. Notice that the lower case version of delta, δ, may also be used to signify a change in a quantity as well as its upper case counterpart, Δ. These changes should then be added to the original values in the first three columns to obtain the new values of the weights and threshold that appear in the next row. Thus, in order to find the weight after all four vectors have been presented, the weight changes in the last row should be added to the weights in the fourth row to give w_1=0. 25, w_2=0.4, θ=0.3.

4.4 The perception

This is an enhancement of the TLU introduced by Rosenblatt (Rosenblatt 1962) and is shown in Figure 4.5. It consists of a TLU whose inputs come from a set of preprocessing *association units* or simply A-units. The input pattern is supposed to be Boolean, that is a set of 1s and 0s, and the A-units can be assigned any arbitrary Boolean functionality but are fixed—they do not learn. The depiction of the input pattern as a grid carries the suggestion that the input may be derived from a visual image, which is the subject of Section 4.6. The rest of the node functions just like a TLU and may therefore be trained in exactly the same way. The TLU may be thought of as a special case of the perception with a trivial set of A-units, each consisting of a single direct connection to one of the inputs. Indeed, sometimes the term "perceptron" is used to mean what we have defined as a TLU. However, whereas a perceptron always performs a linear separation with respect to the output of its A-units, its function of the input space may not be linearly separable if the A-units are non-trivial.

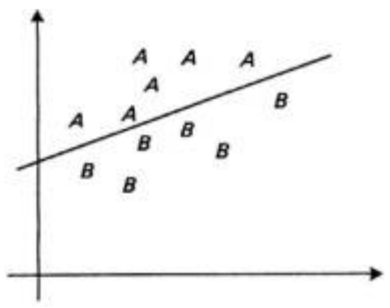


Figure 4.6 Classification of two classes *A, B*.

4.5 Multiple nodes and layers

4.5.1 Single-layer nets

Using the perception training algorithm, we are now in a position to use a single perception or TLU to classify two linearly separable classes *A* and *B*. Although the patterns may have many inputs, we may illustrate the pattern space in a schematic or cartoon way as shown in Figure 4.6. Thus the two axes are not labelled, since they do not correspond to specific vector components, but are merely indicative that we are thinking of the vectors in their pattern space.

It is possible, however, to train multiple nodes on the input space to achieve a set of linearly separable dichotomies of the type shown in Figure 4.6. This might occur, for example, if we wish to classify handwritten alphabetic characters where 26 dichotomies are required, each one separating one letter class from the rest of the alphabet—"A"s from non-" A"s, "B"s from non-"B"s, etc. The entire collection of nodes forms a *single-layer net* as shown in Figure 4.7. Of course, whether each of the above dichotomies is linearly separable is another question. If they are, then the perceptron rule may be applied successfully to each node individually.

4.5.2 Nonlinearly separable classes

Suppose now that there are four classes *A, B, C, D* and that they are separable by two planes in pattern space as shown in Figure 4.8. Once again, this diagram is a schematic representation of a high-dimensional space. It would be futile trying to use a single-layer net to separate these classes since class *A,* for example, is not linearly separable from the others taken together. However, although the problem (identifying the four classes *A, B, C, D*) is not linearly separable, it is possible to solve it by "chopping" the pattern space into linearly separable regions and looking for particular combinations of overlap within these regions. The initial process of pattern space division may be accomplished with a first layer of TLUs and the combinations evaluated by a subsequent layer. This strategy is now explained in detail.

We start by noting that, although no single class (such as *A,* for example) is linearly separable from the others, the higher order class consisting of *A* and *B* together is linearly separable from that consisting of *C* and *D* together. To facilitate talking about these classes, let *AB* be the class consisting of all patterns that are in *A* or *B*. Similarly, let *CD* be the class containing all patterns in C or *D,* etc. Then our observation is that *AB* and *CD* are linearly separable as are *AD* and *BC*.

We may now train two units U_1, U_2 with outputs y_1, y_2 to perform these two dichotomies as shown in Table 4.2.

Table 4.2 y_1, y_2 outputs.

Class	y_1	Class	y_2
AB	1	*AD*	1
CD	0	*BC*	0

Suppose now a member of the original class *A* is input to each of U_1, U_2. From the table, this results in outputs $y_1=y_2=1$. Conversely, suppose an unknown vector **x** is input and both outputs are 1. As far as U_1 is concerned **x** is in *AB,* and U_2 classifies it as in *AD*. The only way it can be in both is if it is in *A*. We conclude therefore that $y_1=1$ and $y_2=1$ if, and only if,

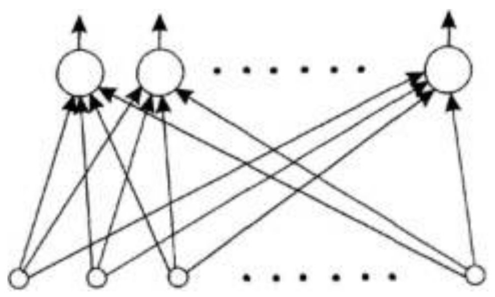


Figure 4.7 Single-layer net.

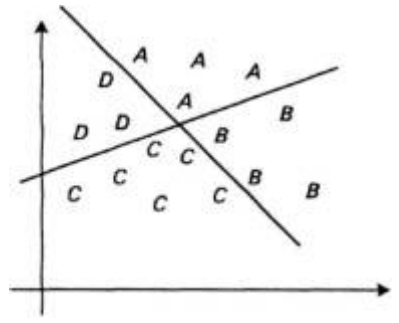


Figure 4.8 Pattern space for classification of four classes *A, B, C, D*.

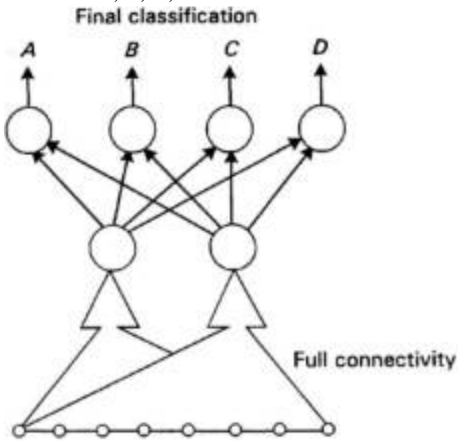


Figure 4.9 Two-layer net for four-class classification

the input vector **x** is in *A*. Proceeding with the other three possibilities, we obtain a unique code, in terms of y_1, y_2, for each of the classes, as shown in Table 4.3.

Table 4.3 y_1, y_2 codes.

y_1	y_2	Class
0	0	*C*
0	1	*D*
1	0	*B*
1	1	*A*

These codes may now be decoded by a set of four two-input TLUs, each connected to both *U1* and *U2* as shown in Figure 4.9. Thus, to signal class *A* we construct a two-input TLU that has output "1" for input (1, 1) and output "0" for all other inputs. To signal class *B* the TLU must output "1" only when presented with (1, 0), and so on for *C* and *D*. These input-output relations are certainly linearly separable since they each consist, in pattern space, of a line that "cuts away" one of the corners of the square (refer back to Fig. 3.4 for an example that corresponds to the A-class node). Notice that only one of the four TLU *output units* is "on" (output "1") at any one time so that the classification is signalled in an unambiguous way.

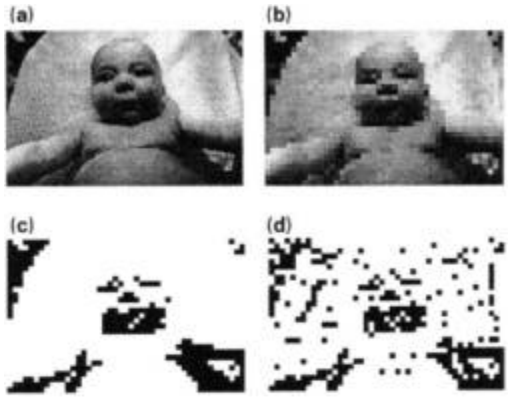


Figure 4.10 Making training sets from images.

Two important points need to be made here. First, the output units were not trained; each one has been assigned the appropriate weights by inspection of their pattern space. Secondly, if we had chosen to use the groupings *AC* or *DB* then we would have failed, since neither of these can take part in a linearly separable dichotomy. There were therefore two pieces of information required in order to train the two units.

(a) The four classes may be separated by two hyperplanes.
(b) *AB* was linearly separable from *CD* and *AD* was linearly separable from *BC*.

It would be more satisfactory if we could dispense with (b) and train the entire two-layer architecture in Figure 4.9 as a whole *ab initio*. The less information we have to supply ourselves, the more useful a network is going to be. In order to do this, it is necessary to introduce a new training algorithm based on a different approach, which obviates the need to have prior knowledge of the pattern space.

Incidentally, there is sometimes disagreement in the literature as to whether the network in Figure 4.9 is a two- or three-layer net. Most authors (as I do) would call it a two-layer net because there are two layers of artificial neurons, which is equivalent to saying there are two layers of weights. Some authors, however, consider the first layer of input distribution points as units in their own right, but since they have no functionality it does not seem appropriate to place them on the same footing as the TLU nodes.

4.6 Some practical matters

We have spoken rather glibly so far about training sets without saying how they may originate in real applications. The training algorithm has also been introduced in the abstract with little heed being paid to how it, and the network, are implemented. This is a suitable point to take time out from the theoretical development and address these issues.

4.6.1 Making training sets

We will make this concrete by way of an example which assumes a network that is being used to classify visual images. The sequence of events for making a single training pattern in this case is shown in Figure 4.10.

Part (a) shows the original scene in monochrome. Colour information adds another level of complexity and the image shown here was, in fact, obtained by first converting from a colour picture. Our goal is somehow to represent this image in a way suitable for input to a TLU or perceptron. The first step is shown in part (b) in which the image has been divided into a series of small squares in a grid-like fashion. Within each square, the luminance intensity is averaged to produce a single grey level. Thus, if a square is located in a region that is mainly dark, it will contain a uniform dark grey, whereas squares in lighter regions will contain uniform areas of pale grey. Each square is called a *pixel* and may now be assigned a number, based on the darkness or lightness of its grey content. One popular scheme divides or *quantizes* the grey-scale into 256 discrete levels and assigns 0 to black and 255 to white. In making the assignment of scale values to pixels, we have to take the value closest to the pixel's grey level. This will result in small *quantization errors,* which will not be too large, however, if there are enough levels to choose from.

If we know how many pixels there are along each side of the picture the rows (or columns) of the grid may now be concatenated to obtain a vector of numbers. This is adequate as it stands for input to a TLU, which does not necessarily need Boolean vectors, but is not suitable for the perceptron. To convert to a Boolean vector we must use only two values of grey, which may be taken to be black and white. This conversion is accomplished by thresholding at some given grey level. For example, if we set the threshold at 50 per cent and are using the 0–255 labelling scheme, all pixels with values between 0 and 127 will be assigned the value 0 (white), while all those between 128 and 255 will be given the label 1 (black). This has been done in part (c) of the figure, which shows the *binarized* version of the image with threshold 50 per cent. The rows (or columns) may now be concatenated to produce a Boolean vector suitable for use as input to the perceptron. Another way of thinking of the binarized image is that it is a direct result of grey-level quantization but with only two (instead of 256) grey levels. Image vectors like those in Figure 4.10b, c may be stored for later use in computer memory or *framestore,* or on disk in a file.

Before leaving our example, we can use it to help illustrate a typical task that our network may be expected to perform. In Figure 4.10d is shown a copy of the original binarized image of part (c) but with some of the pixels having their values inverted. This may have occurred, for example, because the image became corrupted by noise when it was transmitted from a source to a destination machine. Alternatively, we might imagine a more structured alteration in which, for example, the child has moved slightly or has changed facial expression. We would expect a well-trained network to be able to classify these slightly altered images along with the original, which is what we mean by its ability to generalize from the training set.

4.6.2
Real and virtual networks

When we build a neural network do we go to our local electronic hardware store, buy components and then assemble them? The answer, in most cases, is "no". Usually we simulate the network on a conventional computer such as a PC or workstation. What is simulation? The entries in Table 4.1 could have been filled out by pencil and paper calculation, by using a spreadsheet, or by writing a special purpose computer program. All these are examples of simulations of the TLU, although the first method is rather slow and is not advised for general use. In the parlance of computer science, when the net is being simulated on a general purpose computer, it is said to exist as a *virtual machine* (Tanenbaum 1990). The term "virtual reality" has been appropriated for describing simulations of spatial environments—however, the virtual machines came first.

Instead of writing a computer program from scratch, one alternative is to use a general purpose neural network simulator that allows network types and algorithms to be chosen from a set of predetermined options. It is also often the case that they include a set of visualization tools that allow one to monitor the behaviour of the net as it adapts to the training set. This can be extremely important in understanding the development and behaviour of the network; a machine in which the information is distributed in a set of weights can be hard to understand. Examples of this type of simulator are available both commercially and as freely distributed software that may be downloaded via an Internet link. For a survey, see Murre (1995).

Large neural networks can often require many thousands of iterations of their training algorithm to converge on a solution, so that simulation can take a long time. The option, wherever possible, should be to use the most powerful computer available and to limit the network to a size commensurate with the available computing resources. For example, in deciding how large each pixel should be in Figure 4.10, we have to be careful that the resulting vector is not so large that there are too many weights to deal with in a reasonable time at each iteration of the learning algorithm.

In Chapter 1, one of the features of networks that was alluded to was their ability to compute in parallel. That is, each node may be regarded as a processor that operates independently of, and concurrently with, the others. Clearly, in simulation as a virtual machine, networks cannot operate like this. The computation being performed by any node has to take place to the exclusion of the others and each one must be updated in some predefined sequence. In order to take advantage of the latent parallelism, the network must be realized as a physical machine with separate hardware units for each node and, in doing this, there are two aspects that need attention. First, there needs to be special purpose circuitry for implementing the node functionality, which includes, for example, multiplying weights by inputs, summing these together and a nonlinearity output function. Secondly, there needs to be hardware to execute the learning algorithm. This is usually harder to achieve and many early physical network implementations dealt only with node functionality. However, it is the learning that is computer intensive and so attention has now shifted to the inclusion of special purpose learning hardware.

Distinction should also be made between network hardware accelerators and truly parallel machines. In the former, special circuitry is devoted to executing the node function but only one copy exists so that, although there may be a significant speed-up, the network is still operating as a virtual machine in some way. Intermediate structures are also possible in which there may be several node hardware units, allowing for some parallelism, but not sufficient for an entire network. Another possibility is to make use of a general purpose parallel computer, in which case the node functionality and training may be shared out amongst individual processors. Some accounts of special purpose chips for neural networks may be found in two special issues of the *IEEE Transactions on Neural Nets* (Sánchez-Sinencio & Newcomb 1992a, b).

4.7 Summary

By building on the insights gained using the geometric approach introduced in the last chapter, we have demonstrated how TLU-like nodes (including perceptrons) can adapt their weights (or learn) to classify linearly separable problems. The resulting learning rule is incorporated into a training algorithm that iteratively presents vectors to the net and makes the required changes. The threshold may be adapted on the same basis using a simple trick that makes it appear like a weight. We have seen how nonlinearly separable problems may be solved in principle using a two-layer net, but the method outlined so far relies heavily on prior knowledge and the hand crafting of weights. More satisfactory schemes are the subject of subsequent chapters. Finally, the general idea of a "training vector" was made more concrete with reference to an example in vision, and issues concerning the implementation of networks in software and hardware were discussed.

4.8 Notes

1. The symbol □ is read as "implies".