1: Computers and Symbols versus Nets and Neurons

Kevin Gurney[1]

Dept. Human Sciences, Brunel University
Uxbridge, Middx.

[1]These notes are currently under review for publication by UCL Press Limited in the UK. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author. This statement must be easily visible on the first page of any reproduced copies. I would be happy to receive any comments you might have on this draft; send them to me via electronic mail at Kevin.Gurney@brunel.ac.uk. I am particularly interested in hearing about things that you found difficult to learn or that weren't adequately explained, but I am also interested in hearing about inaccuracies, typos, or any other constructive criticism you might have.

1 Neural net: A preliminary definition

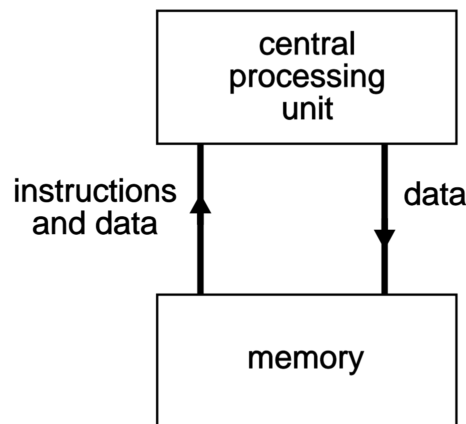
To set the scene it is useful to give a definition of what we mean by 'Neural Net'. However, it is the object of the course to make clear the terms used in this definition and to expand considerably on its content.

> 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 inter-unit connection strengths, or *weights*, obtained by a process of adaptation to, or *learning* from, a set of training patterns.

In order to see how very different this is from the processing done by conventional computers it is worth examining the underlying principles that lie at the heart of all such machines.

2 The von Neumann machine and the symbolic paradigm

The operation of all conventional computers may be modelled in the following way



von-Neumann machine

The computer repeatedly performs the following cycle of events

1. fetch an instruction from memory.
2. fetch any data required by the instruction from memory.
3. execute the instruction (process the data).
4. store results in memory.
5. go back to step 1).

What problems can this method solve easily? It is possible to formalise many problems in terms of an *algorithm*, that is as a well defined procedure or recipe which will guarantee the answer. For example, the solution to a set of equations or the way to search for an item in a database. This algorithm may then be broken down into a set of simpler statements which can, in turn, be reduced eventually, to the instructions that the CPU executes.

In particular, it is possible to process strings of symbols which obey the rules of some formal system and which are interpreted (by humans) as 'ideas' or 'concepts'. It was the hope of the AI programme that all knowledge could be formalised in such a way: that is, it could be reduced to the manipulation of symbols according to rules and this manipulation implemented on a von Neumann machine (conventional computer).

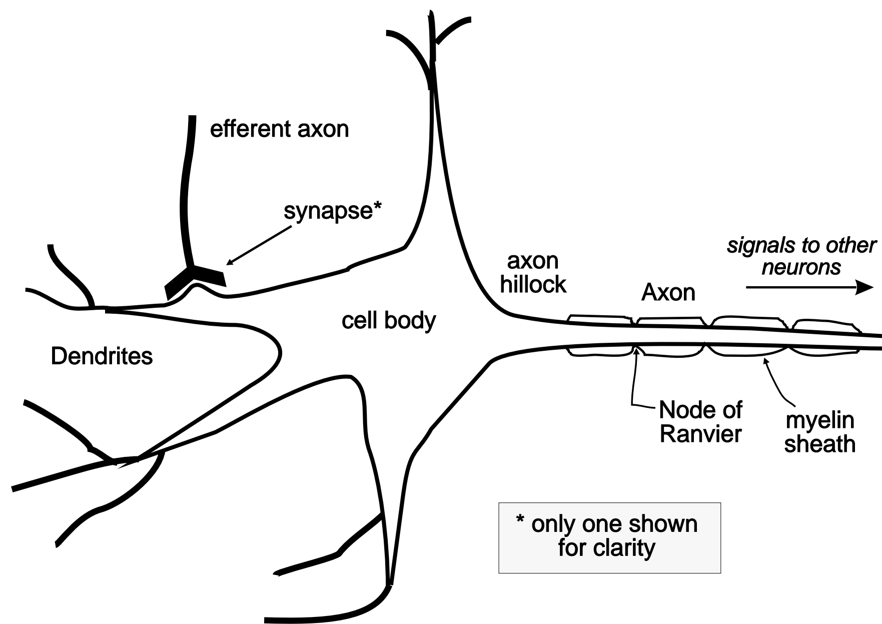
We may draw up a list of the essential characteristics of such machines for comparison with those of networks.

- The machine must be told in advance, and in great detail, the exact series of steps required to perform the algorithm. This series of steps is the *computer program.*
- The type of data it deals with has to be in a precise format - noisy data confuses the machine.
- The hardware is easily degraded - destroy a few key memory locations and the machine will stop functioning or 'crash'.
- There is a clear correspondence between the semantic objects being dealt with (numbers, words, database entries etc) and the machine hardware. Each object can be 'pointed to' in a block of computer memory.

The success of the symbolic approach in AI depends directly on the consequences of the first point above which assumes we can find an algorithm to describe the solution to the problem. It turns out that many everyday tasks we take for granted are difficult to formalise in this way. For example, our visual (or aural) recognition of things in the world; how do we recognise handwritten characters, the particular instances of which, we may never have seen before, or someone's face from an angle we have never encountered? How do we recall whole visual scenes on given some obscure verbal cue? The techniques used in conventional databases are too impoverished to account for the wide diversity of associations we can make.

The way out of these difficulties that shall be explored in this course is that, by copying more closely the physical architecture of the brain, we may emulate brain function more closely.

3 Real neurons: a review



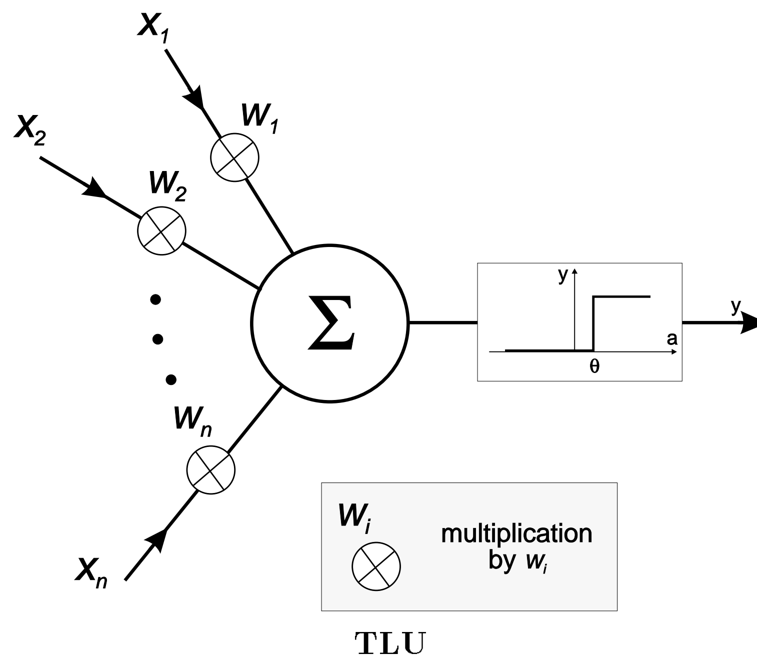
Biological neuron

Signals are transmitted between neurons by electrical pulses (*action-potentials* or 'spike' trains) travelling along the *axon*. These pulses impinge on the afferent neuron at terminals called *synapses*. These are found principally on a set of branching processes emerging from the cell body (*soma*) known as *dendrites*. Each pulse occurring at a synapse initiates the release of a small amount of chemical substance or *neurotransmitter* which travels across the synaptic cleft and which is then received at post-synaptic *receptor* sites on the dendritic side of the synapse. The neurotransmitter becomes bound to molecular sites here which, in turn, initiates a change in the dendritic membrane potential. This *post-synaptic-potential* (PSP) change may serve to increase (*hyperpolarise*) or decrease (*depolarise*) the polarisation of the post-synaptic membrane. In the former case, the PSP tends to inhibit generation of pulses in the afferent neuron, while in the latter, it tends to excite the generation of pulses. The size and type of PSP produced will depend on factors such as the geometry of the synapse and the type of neurotransmitter. Each PSP will travel along its dendrite and spread over the soma, eventually reaching the base of the axon (*axon-hillock*). The afferent neuron sums or integrates the effects of thousands of such PSPs over its dendritic tree and over time. If the integrated potential at the axon-hillock exceeds a threshold, the cell 'fires' and generates an action potential or spike which starts to travel along its axon. This then initiates the whole sequence of events again in neurons contained in the efferent pathway.

4 Artificial neurons: the TLU

The information processing performed in this way may be crudely summarised as follows: signals (action-potentials) appear at the unit's inputs (synapses). The effect (PSP) each signal has may be approximated by multiplying the signal by some number or *weight* to indicate the strength of the synapse. The weighted signals are now summed to produce an overall

unit *activation*. If this activation exceeds a certain threshold the unit produces a an output response. This functionality is captured in the artificial neuron known as the Threshold Logic Unit (TLU) originally proposed by McCulloch and Pitts (McCulloch and Pitts, 1943)



TLU

We suppose there are n inputs with signals $x_1, x_2, \ldots x_n$ and weights $w_1, w_2, \ldots w_n$. The signals take on the values '1' or '0' only. That is the signals are *Boolean* valued. (This allows their relation to digital logic circuits to be discussed). The activation a, is given by

$$a = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \tag{1}$$

This may be represented more compactly as

$$a = \sum_{i=1}^{n} w_i x_i \tag{2}$$

the output y is then given by thresholding the activation

$$y = \begin{cases} 1 & \text{if} \quad a \geq \theta \\ 0 & \text{if} \quad a < \theta \end{cases} \tag{3}$$

The threshold θ will often be zero. The threshold function is sometimes called a *step-function* or *hard-limiter* for obvious reasons. If we are to push the analogy with real neurons, the presence of an action-potential is denoted by binary '1' and its absence by binary '0'.

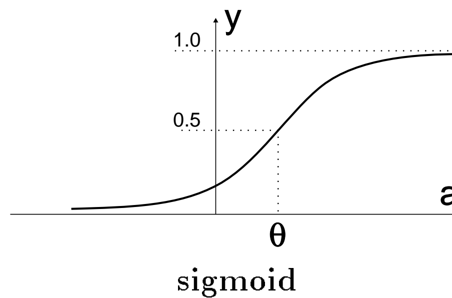
Notice that there is no mention of time so far - the unit responds instantaneously to its input whereas real neurons integrate over time as well as space; how this may be this may be remedied will be discussed later.

5 Non-binary signal communication

The signals dealt with so far (for both real and artificial neurons) take on only two values, that is they are *binary* signals. In the case of real neurons the two values are the action-potential

voltage and the axon membrane resting potential. For the TLUs these were conveniently labelled '1' and '0' respectively. Now, it is generally accepted that, in real neurons, information is encoded in terms of the *frequency* of firing rather than merely the presence or absence of a pulse. (Phase information may also be important but the nature of this mechanism is less certain and will not be discussed here).

There are two ways we can represent this in our artificial neurons. First, we may extend the signal range to be positive real numbers. This works fine at the input straight away, but the use of a step function limits the output signals to be binary. This may be overcome by 'softening' the step-function to a continuous 'squashing' function



sigmoid

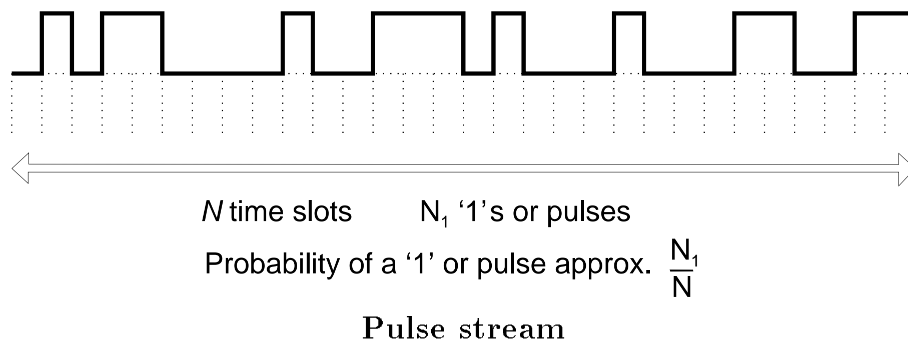
One convenient mathematical form for this is the *sigmoid*

$$y = \sigma(a) \equiv \frac{1}{1 + e^{-a/\rho}} \tag{4}$$

Here, ρ determines the shape of the sigmoid: a larger value making the curve flatter. In many texts, this parameter is omitted so that it is implicitly assigned the value 1. The activation is still given by eqn. (1) but now the output is given by (4). Units with this functionality are sometimes called *semilinear* units. The threshold now corresponds to the activation which gives $y = 0.5$. In (4) the threshold is zero and if we require a non-zero threshold then it must be included by writing

$$y = \sigma(a) \equiv \frac{1}{1 + e^{-(a-\theta)/\rho}} \tag{5}$$

As an alternative to using real (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.

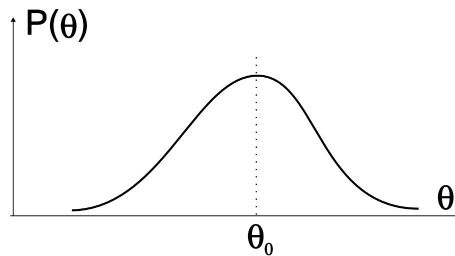


Pulse stream

Time is divided into discrete 'slots'. If the signal level we require is p, where $0 \leq p \leq 1$, then the probability of a '1' appearing at each time slot will be p (If we require values in some other range then just normalise the signal first to the unit interval). The output y, is now

interpreted as the probability of outputting a '1' rather than directly as an analogue signal value. Such units are sometimes known as *stochastic semilinear* units. If we don't know p an estimate may be made by counting the number of '1's, N_1, in N time slots. The probability estimate p^* is given by $p^* = N_1/N$.

In the stochastic case it is possible to reinterpret the sigmoid in a more direct way. First note that it is an approximation to the cumulative gaussian (normal distribution, cf z-scores in statistics). If we had, in fact used the latter then this is equivalent to modelling a 'noisy' threshold; that is the threshold at any time is a random variable with gaussian (normal) distribution.



Normal distribution

Thus, the probability of firing (outputting a '1') if the activation is a, is just the probability that the threshold is less than a, which is just the cumulative of the gaussian up to this value.

6 Introducing time

The artificial neurons discussed so far all evaluate their activation and output 'instantaneously' - there is no integration of signals over time. To introduce dependence on time, we define the activation implicitly by its rate of change da/dt. This requires that the weighted sum of inputs be denoted by some other quantity. Thus, put

$$s = \sum_{i=1}^{n} w_i x_i \tag{6}$$

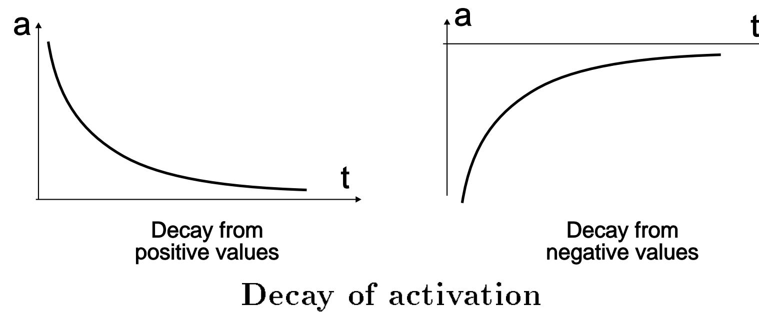
and now put

$$\frac{da}{dt} = -\alpha a + \beta s \tag{7}$$

where α and β are positive constants. The first term gives rise to activation *decay*, while the second represents input from the other neurons and may be excitatory. To see the effect of the decay term put $s = 0$. There are now two cases.

i) $a > 0$. Then $da/dt < 0$; that is, a decreases.

ii) $a < 0$. Then $da/dt > 0$; that is, a increases



Decay of activation

The neuron will reach equilibrium when $da/dt = 0$. That is when

$$a = \frac{\beta s}{a} \tag{8}$$

There are several possible modifications to (7). e.g., 'PDP' vol 3 ch 2, and (von der Malsburg, 1973; Hopfield and Tank, 1986). Stephen Grossberg has made extensive use of this type of functionality - see for example (Grossberg, 1976).

7 Network features

Having looked at the component processors of artificial neural nets, it is time to compare the features of such nets with the von Neumann paradigm. The validity of some of these properties should become clearer as the operation of specific nets is described.

- Clearly the style of processing is completely different - it is more akin to signal processing than symbol processing. The combining of signals and producing new ones is to be contrasted with the execution of instructions stored in a memory.
- Information is stored in a set of weights rather than a program. The weights are supposed to adapt when the net is shown examples from a training set.
- Nets are robust in the presence of noise: small changes in an input signal will not drastically affect a node's output.
- Nets are robust in the presence of hardware failure: a change in a weight may only affect the output for a few of the possible input patterns.
- High level concepts will be represented as a pattern of activity across many nodes rather than as the contents of a small portion of computer memory.
- The net can deal with 'unseen' patterns and generalise from the training set.
- Nets are good at 'perceptual' tasks and associative recall. These are just the tasks that the symbolic approach has difficulties with.

References

Grossberg, S. (1976). Adaptive pattern classification and universal recoding: 1. parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:121 – 134.

Hopfield, J. J. and Tank, D. W. (1986). Computing with neural circuits: A model. *Science*, 233:625 – 633.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7:115 – 133.

von der Malsburg, C. (1973). Self-organisation of orientation sensitive cells in the striate cortex. *Kybernetik*, 14:85 – 100.

2: TLUs and vectors - simple learning rules

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

1 Geometric interpretation of TLU action

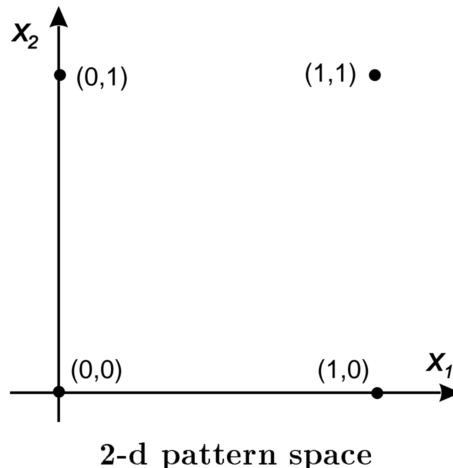
It is possible to interpret the functionality of a TLU geometrically. In summary, it separates its input space into two parts divided by a hyperplane according to whether the input is classified as a '1' or a '0'. We now introduce the ideas contained here step by step.

1.1 Pattern classification and input space

Consider the TLU with weights $w_1 = 1, w_2 = 1$ and threshold 1.5. The table of responses is shown below.

x_1	x_2	activation	output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

The TLU may be thought of as *classifying* its input patterns into two classes: 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 two-dimensional space



2-d pattern space

The space in which the inputs reside is referred to as the *pattern space*. Each pattern determines a point in the space by using its component values as space-coordinates. 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: we shall return to the idea of using higher dimensional spaces later. However, the geometric insight obtained in 2-D will carry over (when expressed algebraically) into n-D.

1.2 The linear separation of classes

Since the critical condition for classification occurs when the activation equals the threshold, it is useful to examine the geometric implication of this. Putting the activation equal to the threshold gives

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

In the 2-D case we are considering

$$w_1 x_1 + w_2 x_2 = \theta \tag{2}$$

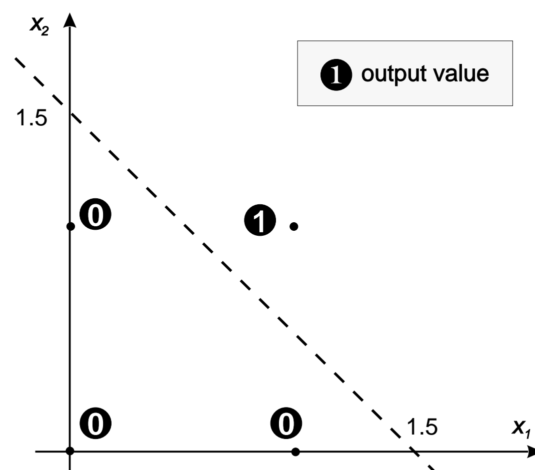
so that

$$x_2 = -\left(\frac{w_1}{w_2}\right) x_1 + \left(\frac{\theta}{w_2}\right) \tag{3}$$

This is of the form

$$x_2 = a x_1 + b \tag{4}$$

That is, a straight line with slope a and intercept b on the x_2 axis. Since $w_1 = w_2 = 1$ and $a = -1$, we also have $b = 1.5$.



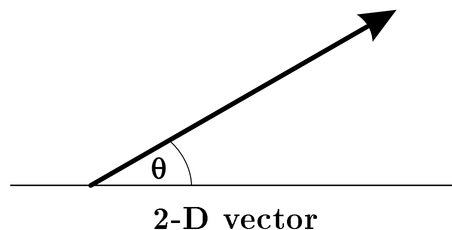
Line in 2-d pattern space

The two classes are therefore separated by the 'decision' line which is defined by putting the activation equal to the threshold. It turns out that it is possible to generalise this result to TLUs with n inputs In 3-D the two classes are separated by a *decision-plane*. In n-D this becomes a *decision-hyperplane*. (The 'hyper-' is sometimes dropped even when $n > 3$). The

converse of this is that, any TLU is defined by some hyperplane in its pattern space and any function which cannot be realised in this way cannot be realised by a TLU. Because the defining equation (1) for the hyperplane, is *linear*, the TLU is a *linear classifier*. Using some ideas about *vectors*, it is possible to prove these results and to gain insight into what is going on.

2 Vectors

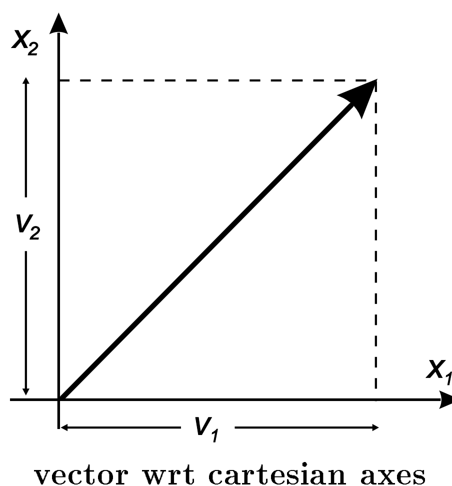
Vectors* are usually introduced as representations of quantities that have magnitude and direction. Thus the velocity of a car is defined by the car's speed and direction. On paper we may draw an arrow whose length is proportional to the speed and whose direction is the same as that of the car.



2-D vector

Vectors will be denoted by bold face letters e.g. $\mathbf{v}$. The magnitude of $\mathbf{v}$ will be denoted by $\|\mathbf{v}\|$. In writing vectors we can't use bold so we usually put an underline thus – $\underline{v}$. The length of vectors is sometimes denoted by the italic letter e.g. v. In accordance with our geometric ideas a vector is now defined by the pair of numbers $(\|\mathbf{v}\|, \theta)$ where θ is the angle the vector makes with some reference direction.

In order to generalise to higher dimensions, and to relate vectors to the ideas of pattern space, it is more convenient to describe vectors with respect to a cartesian coordinate system. That is, we give the projected lengths of the vector onto two perpendicular axes



vector wrt cartesian axes

The vector is now described by the pair of numbers v_1, v_2. These numbers are its *components* in the chosen coordinate 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

*For a good introduction to vectors as required for neural nets, see chapter 9 in PDP vol. 1

vector is now an ordered list of numbers. Notice the ordering is important, since (1,3) is in a different direction from (3,1).

This definition immediately generalises to more than 2-dimensions. An n-dimensional vector is simply an ordered list of n numbers, $\mathbf{v} = (v_1, v_2, \ldots v_n)$. Lists like this appeared on the first problem sheet. They were the *weight vector* $(w_1, w_2, \ldots, w_n)$ and the *input vector* $(x_1, x_2, \ldots x_n)$ for the node.

2.1 The length of a vector

For our 2-D prototype, the length of a vector is just its length in the plane. In terms of its components, this is given by pythagoras's theorem.

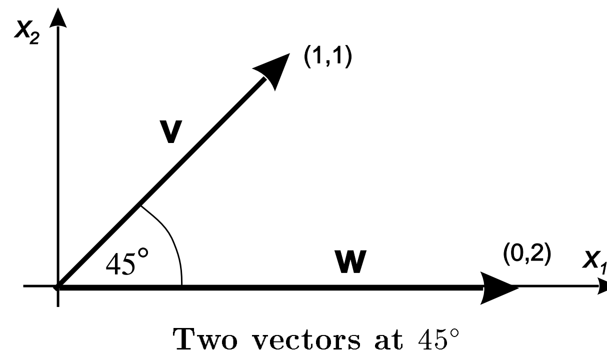
$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2} \tag{5}$$

In n-dimensions, the length is *defined* by the natural extension of (5)

$$\|\mathbf{v}\| = \left[\sum_{i=1}^{n} v_i^2\right]^{\frac{1}{2}} \tag{6}$$

2.2 Comparing vectors - the inner product

Consider the vectors $\mathbf{v} = (1, 1)$ and $\mathbf{w} = (0, 2)$ shown below



Two vectors at $45°$

The angle between them is $45°$. Define the *inner product* $\mathbf{v} \cdot \mathbf{w}$ of the two vectors by

$$\mathbf{v} \cdot \mathbf{w} = v_1 w_1 + v_2 w_2 \tag{7}$$

The form on the right-hand side should be familiar - it is just the same as that we have used to define the activation of a TLU... Substituting the component values gives $\mathbf{v} \cdot \mathbf{w} = 2$. We will now try and give this a geometric interpretation.

First we note that $\|\mathbf{v}\| = \sqrt{2}$ and $\|\mathbf{w}\| = 2$. Next we observe that the *cosine* of the angle between the vectors is $1/\sqrt{2}$

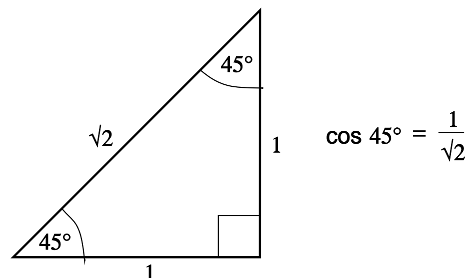


diagram of cos 45

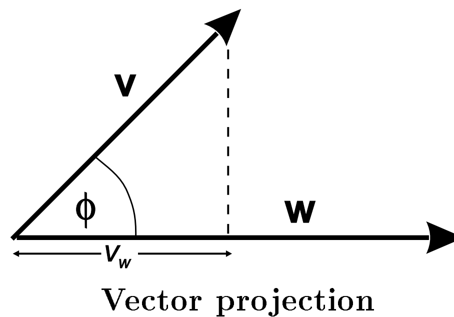
Therefore $\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\|\|\mathbf{w}\| \cos 45°$. It may be shown that this is a general result; that is, if the angle between two vectors $\mathbf{v}$ and $\mathbf{w}$ is ϕ then the definition of dot-product given in (7) is equivalent to

$$\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\|\|\mathbf{w}\| \cos \phi \tag{8}$$

(For a proof of this see ch 9 PDP vol 1). In n-dimensions the inner product is defined by the natural extension of (7)

$$\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{n} w_i v_i \tag{9}$$

Although we cannot now draw the vectors (for $n > 3$) the geometric insight we obtained for 2-D may be carried over since the behaviour must be the same (the definition is the same). We therefore now examine the significance of the inner product in 2 dimensions. Essentially the inner product tells us how well 'aligned' two vectors are. To see this, let v_w be the component of $\mathbf{v}$ along the direction of $\mathbf{w}$, or the *projection* of $\mathbf{v}$ along $\mathbf{w}$.



Vector projection

The projection is given by $\|\mathbf{v}\| \cos \phi$ which, by (8) gives us

$$v_w = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{w}\|} \tag{10}$$

If ϕ is small then $\cos \phi$ is close to its maximal value of one. The inner product, and hence the projection v_w, will be close to its maximal value; the vectors 'line up' quite well. If $\phi = 90°$, the cosine term is zero and so too is the inner product. The projection is zero because the vectors are at right angles; they are said to be *orthogonal.* If $90 < \phi < 270$, the cosine is negative and so too, therefore, is the projection; its magnitude may be large but the negative sign indicates that the two vectors point into opposite half-planes.

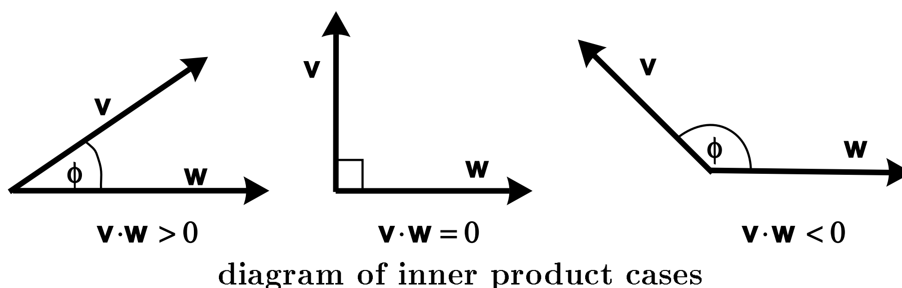


diagram of inner product cases

2.3 Inner products and TLUs

Using the ideas developed above we may express the action of a TLU in terms of the weight and input vectors. The activation a may now be expressed as

$$a = \mathbf{w} \cdot \mathbf{x} \tag{11}$$

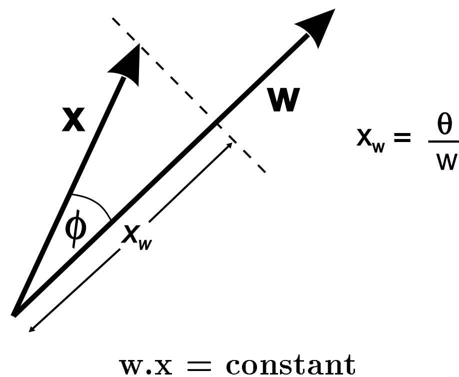
The vector equivalent to (1) now becomes

$$\mathbf{w} \cdot \mathbf{x} = \theta \tag{12}$$

If $\mathbf{w}$ and θ are constant then this implies the projection x_w, of $\mathbf{x}$ along $\mathbf{w}$ is constant since

$$x_w = \frac{\theta}{\|\mathbf{w}\|} \tag{13}$$

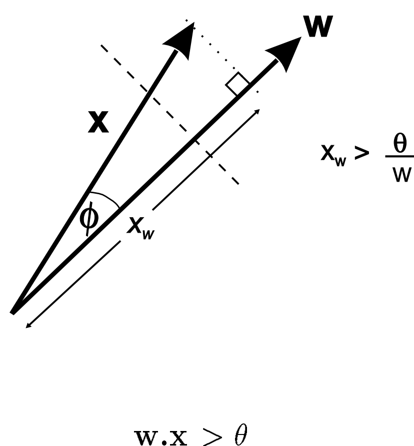
In 2-D, therefore, the equation specifies all $\mathbf{x}$ which lie on a line perpendicular to $\mathbf{w}$.



w.x = constant

There are now two cases

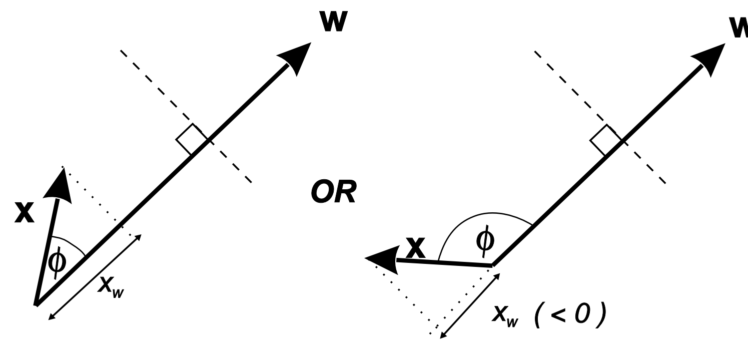
1. If $x_w > \theta/\|\mathbf{w}\|$, then $\mathbf{x}$ must lie beyond the dotted line



$\mathbf{w.x} > \theta$

Using (10) we have that $x_w > \theta/\|\mathbf{w}\|$ implies $\mathbf{w} \cdot \mathbf{x} > \theta$; that is, the activation is greater than the threshold

2. Conversely, if $x_w < \theta$ then $\mathbf{x}$ must lie *below* the dotted line



$$x_w < \frac{\theta}{w}$$

$$\mathbf{w.x} < \theta$$

Using (10) we have that $x_w < \theta/\|\mathbf{w}\|$ implies $\mathbf{w} \cdot \mathbf{x} < \theta$; that is, the activation is less than the threshold

In 3-D the line becomes a plane intersecting the cube and in n-D a hyperplane intersecting the n-dimensional hypercube or n-cube.

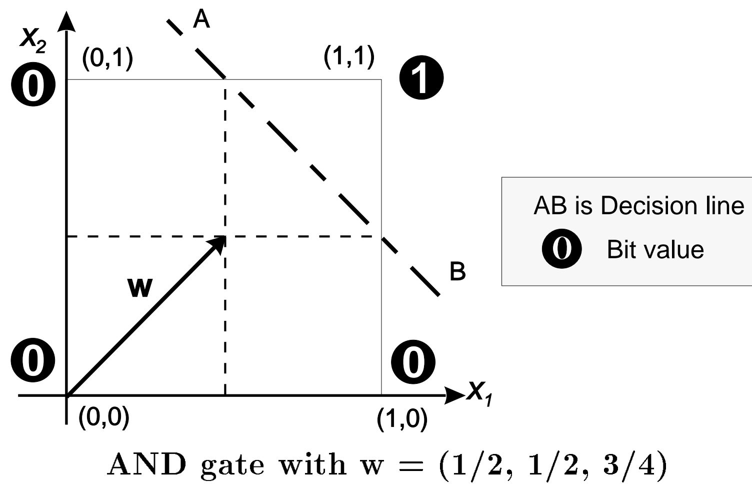
3 Training TLUs

Training any unit consists of adjusting the weight vector and threshold so that the desired classification is performed. In order to place the adaptation of the threshold on the same footing as the weights, we suppose that it is another weight which is permanently connected to an input of -1. I 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

$$\begin{array}{lcl} \mathbf{w} \cdot \mathbf{x} \geq 0 & \Rightarrow & y = 1 \\ \mathbf{w} \cdot \mathbf{x} < 0 & \Rightarrow & y = 0 \end{array} \tag{14}$$

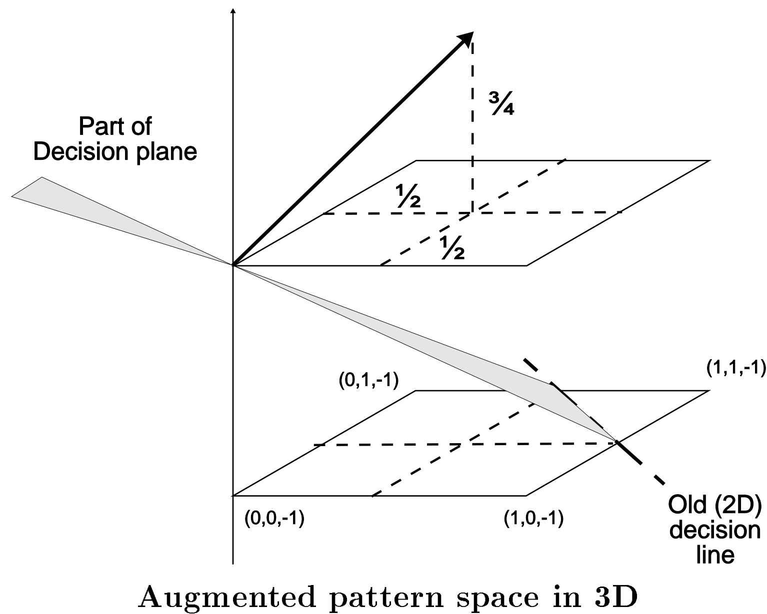
Putting $\mathbf{w} \cdot \mathbf{x} = 0$ now defines the decision (hyper)plane. This plane is therefore orthogonal to the (augmented) weight vector and passes through the origin of the (augmented) pattern space.

To see this consider the example below



AND gate with w = (1/2, 1/2, 3/4)

The result in the augmented space is shown below

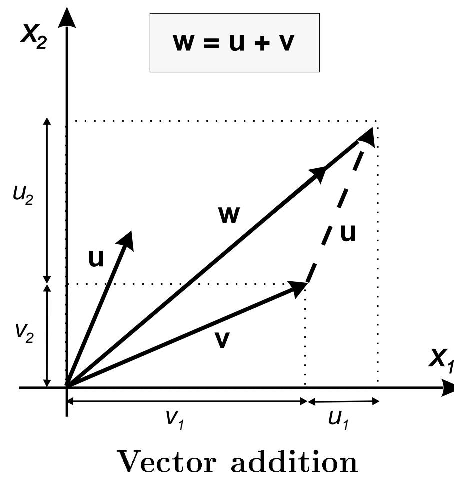


Augmented pattern space in 3D

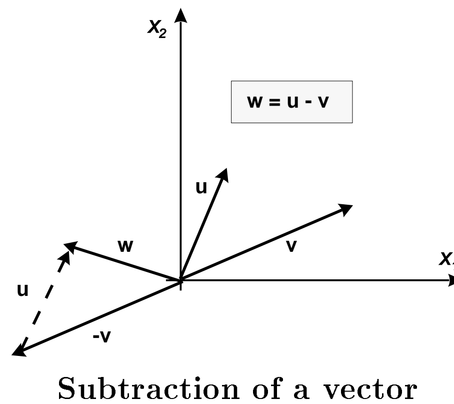
Changing any vector (and hence the weight vector in particular) may be thought of as adding another vector to it. Two vectors $\mathbf{u} = (u_1, u_2)$, $\mathbf{v} = (v_1, v_2)$ may be added by summing their components to give a new vector $\mathbf{w} = (w_1, w_2)$ where

$$\begin{aligned} w_1 &= u_1 + v_1 \\ w_2 &= u_2 + v2 \end{aligned} \tag{15}$$

Subtraction may be defined by noting that the negative of a vector is just the same vector pointing in the opposite direction. Then subtraction of $\mathbf{v}$ is just addition of $-\mathbf{v}$. These operations are shown below.



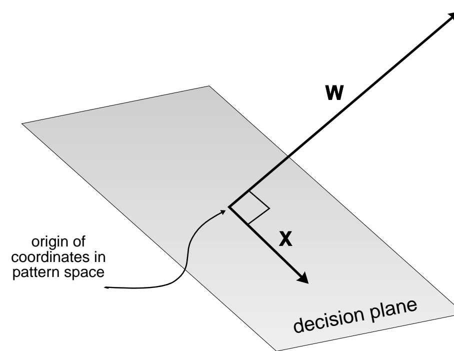
Vector addition



Subtraction of a vector

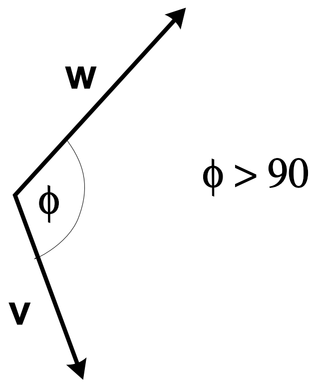
3.1 Adjusting the weight vector

We require the weight vector to be orthogonal to the decision plane and that the plane must pass through the origin.



Orthogonal/origin requirement

The training set for the TLU will consist 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. This type of training is known as *supervised* training, since we tell the net what output is expected for each vector. Suppose we present a training vector $\mathbf{v}$ to the TLU whose current weight vector is $\mathbf{w}$. Further, suppose that $t = 1$ but that, with the weight vector as it is, it produces an output of $y = 0$. To produce a '0' the activation must have been negative when it should have been positive. The situation is shown below.

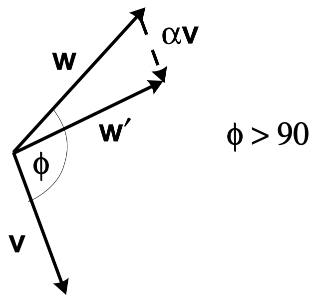


Misclassification 1 - 0

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

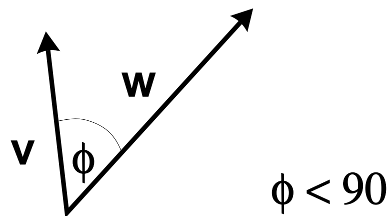
$$\mathbf{w}' = \mathbf{w} + \alpha\mathbf{v} \tag{16}$$

where $0 < \alpha < 1$



increment made to weight vector in case 1

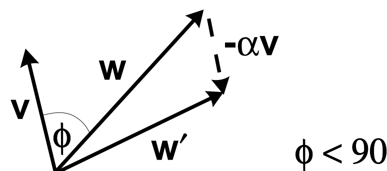
Suppose now instead, that $t = 0$ but $y = 1$. This means the activation was positive when it should have been negative.



Misclassification 0 - 1

We now need to rotate **w** *away* from **v** which may be effected by *subtracting* a fraction of **v** from **w**; that is

$$\mathbf{w}' = \mathbf{w} - \alpha\mathbf{v} \tag{17}$$



Increment made to weight vector in case 2

Both (16) and (17) may be written as a single rule as follows

$$\mathbf{w}' = \mathbf{w} + \alpha(t - y)\mathbf{v} \tag{18}$$

This may be written in terms of the change in the weight vector† $\Delta\mathbf{w}$ or in terms of the components

$$\Delta w_i = \alpha(t - y)v_i \tag{19}$$

This is our first example of a *training rule* or *learning rule*. The parameter α is called the *learning rate*. The learning rule may be incorporated into a *training algorithm* for TLUs as follows

```
repeat
    for each training vector pair (v, t)
        evaluate the output y when v is input to the TLU
        if y ≠ t then
            form a new weight vector w' according to (18)
        else
            do nothing
        end if
    end for
until y = t for all vectors
```

This is usually known as the *Perceptron learning algorithm* because it was used extensively with an extension of the TLU known as the perceptron described in the next section. We now look at an example of using this method with an ordinary TLU.

3.2 Example

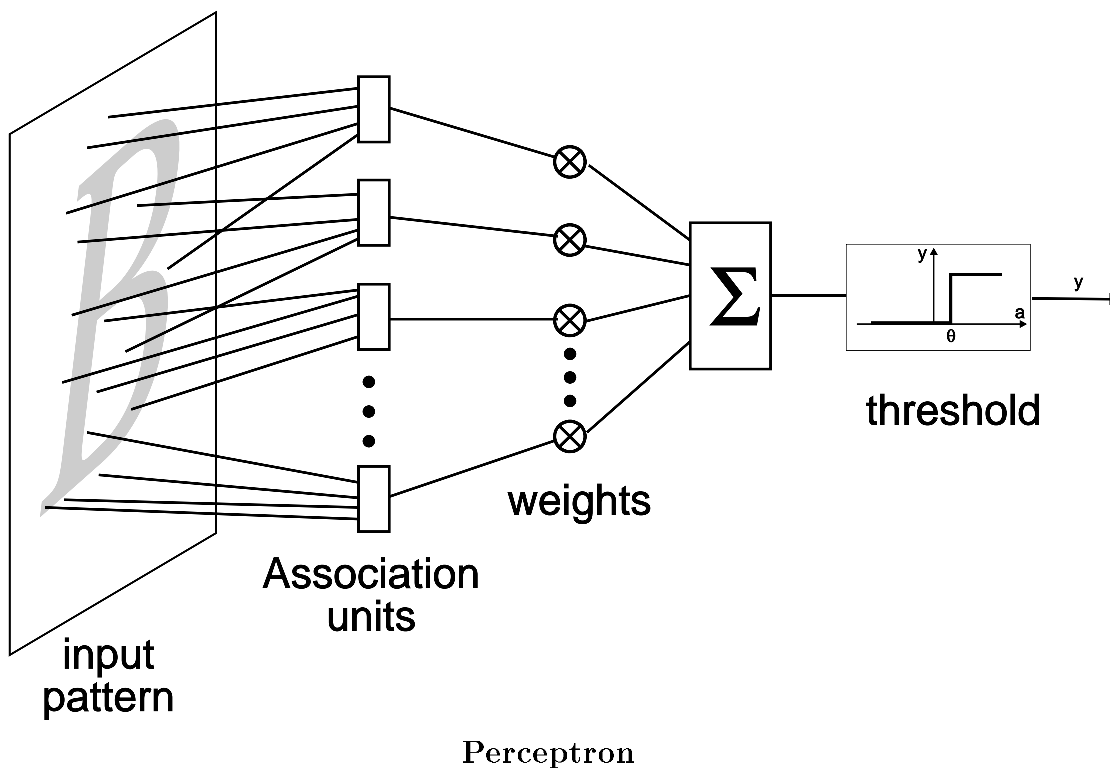
A TLU has weights 0, 0.4 and threshold 0.3. It has to learn the logical AND function; [all outputs '0' except with input (1,1)]. The learning rate is 0.25. Using the above algorithm, the following sequence of events takes place.

4 The Perceptron

This is an enhancement of the TLU introduced by Rosenblatt (Rosenblatt, 1962). It consists of a TLU whose inputs come from a set of preprocessing *association units* (A-units).

†In general, a change in a quantity is denoted by the by the Greek letter 'delta'; either upper case Δ or lower case δ

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
0.25	0.4	0.3	0	0	0	0	0	0	0	0	0
0.25	0.4	0.3	0	1	0.4	1	0	-0.25	0	-0.25	0.25
0.25	0.15	0.55	1	0	0.25	0	0	0	0	0	0
0.25	0.15	0.55	1	1	0.4	0	1	0.25	0.25	0.25	-0.25
0.5	0.4	0.3	0	0	0	0	0	0	0	0	0
0.5	0.4	0.3	0	1	0.4	1	0	-0.25	0	-0.25	0.25
0.5	0.15	0.55	1	0	0.5	0	0	0	0	0	0



Perceptron

The A-units can be assigned any arbitrary Boolean functionality but are fixed - they do not learn. The rest of the node functions just like a TLU and may be therefore be trained in exactly the same way. Rosenblatt was the first to use the training algorithm described here - hence the name 'perceptron training'.

References

Rosenblatt, F. (1962). *Principles of Neurodynamics.* Spartan Books.

3: The delta rule

Kevin Gurney

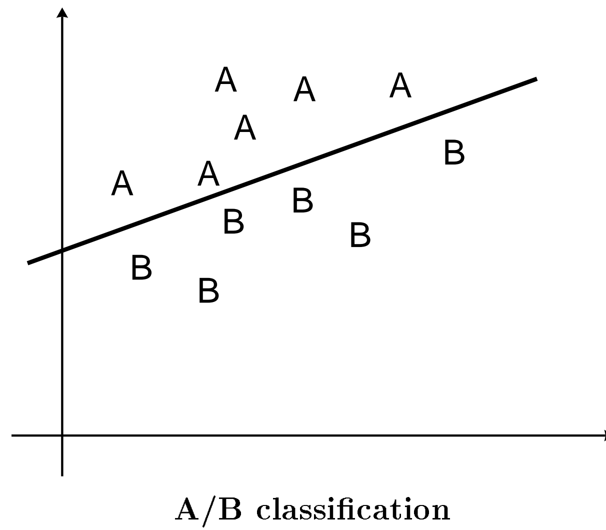
Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

1 Generating input vectors for Neural Nets

In order to make the potential applications discussed subsequently more concrete, we shall consider the example of how image information may be captured and input to a network. Suppose we have a TV camera (monochrome for simplicity) which is viewing a picture that is to be used in training. The output from this is a picture where each point is represented by a continuously variable voltage (analogue quantity) so that shades of grey may be encoded accurately. For a perceptron, however, we require a set of Binary values ('1', '0'). The conversion process is done by dividing the picture into a grid of picture elements or *pixels* each of which is allowed to take only one of two values - black or white. To find the value for each pixel, the average value of the image in the pixel area is found and then thresholded to determine whether it is white or black. We now make the correspondence white = '1', say and black = '0'. This array of Boolean quantities may now be stored in a special purpose computer memory or *framestore*. Typically the pixel grid may be 512 by 512 giving over 1/4 million pixels. Thus, the pattern space will have dimension 1/4 million. This is often reduced to make things more manageable.

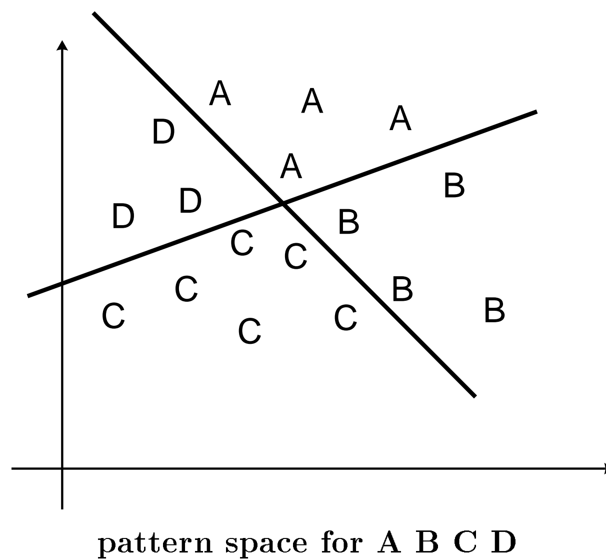
2 Using TLUs and perceptrons as classifiers

Using the perceptron training algorithm, we may now use a perceptron to classify two linearly separable classes A and B. Examples from these classes may have been obtained, for example, by capturing images in a framestore; there may be two classes of faces, or we want to separate handwritten characters into numerals and letters.



A/B classification

Suppose now there are 4 classes *A, B, C, D* and that they are separable by two planes in pattern space



pattern space for A B C D

That is the two classes *(A,B)* *(C,D)* are linearly separable, as too are the classes *(A,D)* and *(B,C)*.

We may now train two units (with outputs y_1, y_2) to perform these two classifications

	1	0
y_1	(A B)	(C D)
y_2	(A D)	(B C)

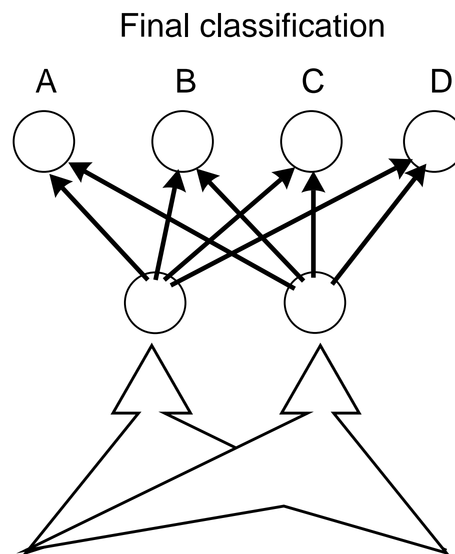
y1 y2 classification

This gives a table encoding the original 4 classes

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

y1 y2 coding for A B C D

The output of the two units may now be decoded by four 2-input TLUs to give the desired responses



2 layer net giving A B C D classification

These *output units* are not trained; each one is assigned weights required to signal a '1' when its class code appears at its inputs. For example, output unit 'A' is the logic AND gate given as an example at the beginning of lecture 2.

Notice that the grouping (A,C) (D,B) would not have worked, since these are not linearly separable, and other arrangements of the four classes in pattern space will require a different set of groupings. There were therefore two pieces of information required in order to train the two units.

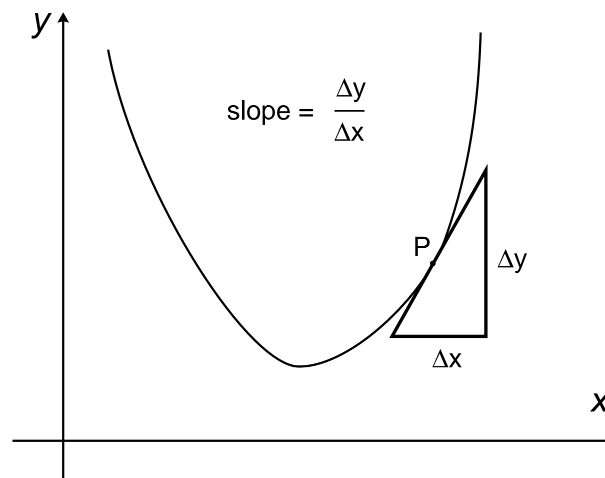
1. The four classes may separated by 2-hyperplanes
2. *(A,B)* was linearly separable from *(C,D)* and *(A,D)* was linearly separable from *(B,C)*.

It would be more satisfactory if we could dispense with 2) and train the entire 2-layer architecture, shown above, as a whole *ab initio*. The less knowledge we have to glean by 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 slightly different approach which obviates the need to know the nature of the nodes' hyperplanes.

3 Minimising an error: the delta rule

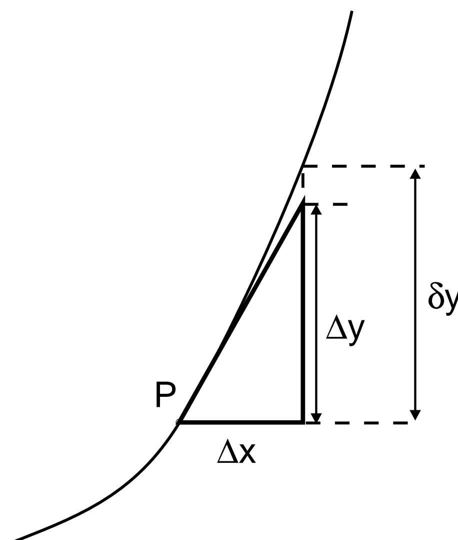
3.1 Finding the minimum of a function: gradient descent

Suppose y is some function of x (y depends on x or $y = y(x)$) but we don't know the exact form of this function. Further, suppose we wish to find the position (x-coordinate) of the minimum value of the function and we can find the slope (rate of change of y) at any point. The slope is just $\Delta y/\Delta x$ in the diagram.



y = y(x) and slope

The slope of a function at any point is the gradient (cf hill gradients) of the tangent to the curve at the point. If Δx is small, then Δy is almost the same as the change δy in the function y, when the change Δx is made in x.



small changes

That is

$$\delta y \approx \Delta y = \frac{\Delta y}{\Delta x}\Delta x \tag{1}$$

so that

$$\delta y \approx \text{slope} \times \Delta x \tag{2}$$

Now put

$$\Delta x = -\alpha \times \text{slope} \tag{3}$$

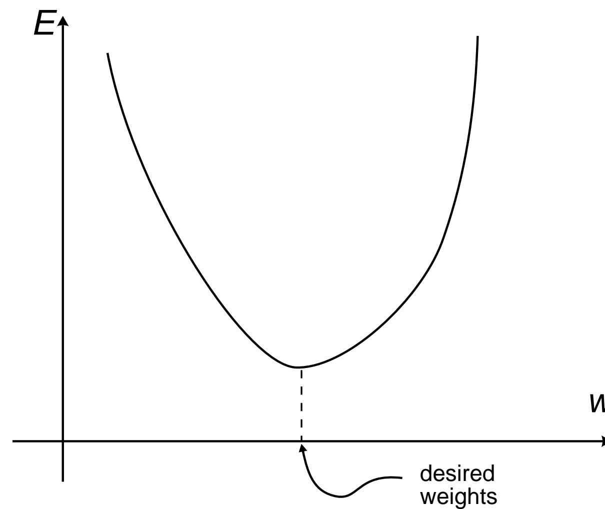
where $\alpha > 0$ and is small enough to ensure $\delta y \approx \Delta y$ then

$$\delta y \approx -\alpha(\text{slope})^2 \tag{4}$$

That is $\delta y < 0$ and we have 'travelled down' the curve towards the minimal point. If we keep repeating steps like (4) iteratively, then we should approach the value of x associated with the function minimum. This technique is called *gradient descent*. How can this be used to train networks?

3.2 gradient descent on an error

The idea is to calculate an error each time the net is presented with a training vector (given that we have supervised learning where there is a target) and to perform a gradient descent on the error considered as function of the weights. There will be a gradient or slope for each weight. Thus, we find the weights which give the minimal error. The situation is as follows.



gradient descent - E vs w

Formally, for each pattern p, we assign an error E_p which is a function of the weights; that is $_p = E_p(w_1, w_2, \ldots, w_n)$. Typically this is defined by the square difference between the output and the target. Thus (for a single node)

$$E_p = \frac{1}{2}(t - y)^2 \tag{5}$$

Where we regard y as a function of the weights. The total error E, is then just the sum of the pattern errors

$$E = \sum_p E_p \tag{6}$$

Now, in order to perform gradient descent, the error must be a continuous function of the weights and there must be a well defined gradient at each point. With TLUs, however, this is not the case; although the *activation* is a continuous function of the weights, the output changes abruptly as the activation passes throught the threshold value.

One way to remedy this is to train on the activation itself rather than the output. This technique is usually ascribed to Widrow and Hoff (Widrow and Hoff, 1960) who trained TLUs which had had their outputs labelled -1, 1 instead of 0, 1. These units they called Adaptive Linear Elements or ADALINEs. For a description of their techniques see (Widrow and Stearns, 1985; Widrow et al., 1987). The learning rule based on gradient decsent with this type of node is, therefore, sometimes known as the Widrow Hoff rule, but more usually now, as the *delta rule*.

We must still supply a target which is the activation the node is supposed to give in response to the training pattern. Recall (lecture 2) that, if the threshold of a TLU is treated as a weight, the condition for classifying as a '1' was that the activation, should be greater (or equal to) zero. Conversely for a '0' to be output we require the activation to be less than zero. We may therefore choose, as our target activations for the two classes, any two numbers of opposite sign. It is convenient to choose the set $\{-1, 1\}$.

The learning rule may now be obtained by finding the slope of the error in (5) with respect to ('wrt') each of the weights, but using activation a rather than output y. That is, for the delta rule with TLUS

$$E_p = \frac{1}{2}(t-a)^2 \tag{7}$$

It may be shown (use of 'function-of-a-function' in calculus) that the slope of E_p with respect to w_j is just $-(t-a)x_j$. The learning rule (delta rule) is now defined by making a change in the weight Δw_j in line with (3)

$$\begin{aligned}\Delta w_j &= -\alpha \times (\text{slope of } E_p \text{ wrt } w_j) \\ &= \alpha(t-a)x_j \end{aligned} \tag{8}$$

This rule may incorporated into a training algorithm similar to the one given in lecture 2. However, the error will never be exactly zero and so the possibility of 'do nothing' given there, will never arise with the delta rule - there will always be some update to the weights. The term $\alpha(t-a)$ is sometimes know as the 'delta' (or δ).

An example of this rule is provided below in which we train the same TLU as used in the Perceptron example of lecture 2 [initial weights (0, 0.4) threshold 0.3, learn rate 0.25].

v	w_1	w_2	θ	x_1	x_2	$a-\theta$	t	δ	δw_1	δw_2	$\delta\theta$	E
1	0.00	0.40	0.30	0	0	-0.30	-1.00	-0.17	-0.00	-0.00	0.17	0.24
2	0.00	0.40	0.48	0	1	-0.08	-1.00	-0.23	-0.00	-0.23	0.23	0.43
3	0.00	0.17	0.71	1	0	-0.71	-1.00	-0.07	-0.07	-0.00	0.07	0.04
4	-0.07	0.17	0.78	1	1	-0.68	1.00	0.42	0.42	0.42	-0.42	1.42
1	0.35	0.59	0.36	0	0	-0.36	-1.00	-0.16	-0.00	-0.00	0.16	0.21
2	0.35	0.59	0.52	0	1	0.07	-1.00	-0.27	-0.00	-0.27	0.27	0.57
3	0.35	0.32	0.79	1	0	-0.44	-1.00	-0.14	-0.14	-0.00	0.14	0.16
4	0.21	0.32	0.93	1	1	-0.40	1.00	0.35	0.35	0.35	-0.35	0.98
1	0.56	0.67	0.58	0	0	-0.58	-1.00	-0.11	-0.00	-0.00	0.11	0.09
2	0.56	0.67	0.68	0	1	-0.01	-1.00	-0.25	-0.00	-0.25	0.25	0.49
3	0.56	0.42	0.93	1	0	-0.37	-1.00	-0.16	-0.16	-0.00	0.16	0.20
4	0.40	0.42	1.09	1	1	-0.26	1.00	0.32	0.32	0.32	-0.32	0.80
1	0.72	0.74	0.77	0	0	-0.77	-1.00	-0.06	-0.00	-0.00	0.06	0.03
2	0.72	0.74	0.83	0	1	-0.09	-1.00	-0.23	-0.00	-0.23	0.23	0.42
3	0.72	0.51	1.06	1	0	-0.34	-1.00	-0.16	-0.16	-0.00	0.16	0.22
4	0.55	0.51	1.22	1	1	-0.16	1.00	0.29	0.29	0.29	-0.29	0.67
1	0.84	0.80	0.93	0	0	-0.93	-1.00	-0.02	-0.00	-0.00	0.02	0.00
2	0.84	0.80	0.95	0	1	-0.15	-1.00	-0.21	-0.00	-0.21	0.21	0.36
3	0.84	0.59	1.16	1	0	-0.32	-1.00	-0.17	-0.17	-0.00	0.17	0.23
4	0.67	0.59	1.33	1	1	-0.07	1.00	0.27	0.27	0.27	-0.27	0.57
1	0.94	0.86	1.06	0	0	-1.06	-1.00	0.02	0.00	0.00	-0.02	0.00
2	0.94	0.86	1.05	0	1	-0.19	-1.00	-0.20	-0.00	-0.20	0.20	0.33
3	0.94	0.65	1.25	1	0	-0.31	-1.00	-0.17	-0.17	-0.00	0.17	0.24
4	0.77	0.65	1.42	1	1	-0.00	1.00	0.25	0.25	0.25	-0.25	0.50
1	1.02	0.90	1.17	0	0	-1.17	-1.00	0.04	0.00	0.00	-0.04	0.01
2	1.02	0.90	1.13	0	1	-0.22	-1.00	-0.19	-0.00	-0.19	0.19	0.30
3	1.02	0.71	1.32	1	0	-0.31	-1.00	-0.17	-0.17	-0.00	0.17	0.24
4	0.84	0.71	1.50	1	1	0.06	1.00	0.24	0.24	0.24	-0.24	0.44

First 'correct pass' through the training set. The following training decreases the error but does not change the classification after thresholding the activationAfter this the

v	w_1	w_2	θ	x_1	x_2	$a-\theta$	t	δ	δw_1	δw_2	$\delta\theta$	E
1	1.08	0.95	1.26	0	0	-1.26	-1.00	0.07	0.00	0.00	-0.07	0.03
2	1.08	0.95	1.20	0	1	-0.25	-1.00	-0.19	-0.00	-0.19	0.19	0.28
3	1.08	0.76	1.38	1	0	-0.30	-1.00	-0.17	-0.17	-0.00	0.17	0.24
4	0.91	0.76	1.56	1	1	0.11	1.00	0.22	0.22	0.22	-0.22	0.40
1	1.13	0.98	1.33	0	0	-1.33	-1.00	0.08	0.00	0.00	-0.08	0.06
2	1.13	0.98	1.25	0	1	-0.27	-1.00	-0.18	-0.00	-0.18	0.18	0.27
3	1.13	0.80	1.43	1	0	-0.30	-1.00	-0.17	-0.17	-0.00	0.17	0.24
4	0.95	0.80	1.61	1	1	0.15	1.00	0.21	0.21	0.21	-0.21	0.36

Examination of (8) shows that it looks formally the same as the perceptron rule [lecture 2]. However, the latter uses the output for comparison with a target, while the delta rule uses the activation. They were also obtained from different theoretical starting points. The perceptron rule was derived by a consideration of hyperplane manipulation while the delta rule is given by gradient descent on the square error.

It was noted above that the discontinuity in error for TLUs could be traced to the discontinuous output function. With semilinear units this is not the case since the sigmoid is a smooth function. Now we may use the error in (5) (using the output rather than the activation) but have to include an extra term which is related to the slope of the sigmoid; that is, the derivative $\sigma'(a)$. So for semilinear units the delta rule becomes

$$\Delta w_j = \alpha \sigma'(a)(t - y)x_j \tag{9}$$

It may be shown that

$$\sigma'(a) \equiv \frac{d\sigma(a)}{da} = \frac{1}{\sigma}\sigma(a)(1 - \sigma(a)) \tag{10}$$

Unlike the perceptron rule, it is possible to generalise the delta rule to train more than a single layer at once. It turns out to be possible to calculate the slope of the error gradient at intermediate network layers. This was our original goal and is fulfilled in the so-called Backpropagation algorithm or generalised delta rule to be dealt with in the next lecture.

References

Widrow, B. and Hoff (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pages 96 – 104. IRE.
Reprinted in *Neurocomputing - Foundations of Research* eds. Anderson and Rosenfeld. This is a third party report on Widrow's paper. It is largely of historic interest only.

Widrow, B. and Stearns, S. (1985). *Adaptive Signal Processing.* Prentice-Hall.
Is in the library short loan section. This is a book on signal processing (Widrow is an engineer) but contains an extensive analysis of gradient descent. The ADALINE stuff is in the first half of the book.

Widrow, B., Winter, and Baxter (1987). Learning phenomena in layered neural networks. In *1st Int. Conference Neural Nets, San Diego*, volume 2, page 411. I have this.This gives a nice description of training linear units and the ideas of linear separability.

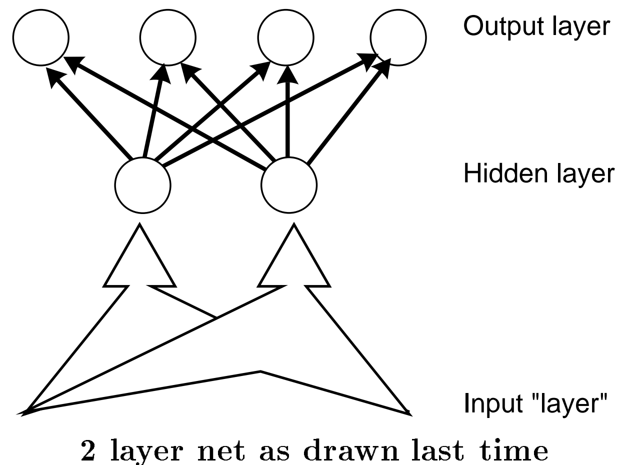
4: Multilayer nets and backpropagation

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

1 Introduction

Recall that the goal we set out to achieve last time was to train a two layer net *in toto* without the clumsy approach of training the intermediate layer separately and then hand-crafting the output layer. This also required prior knowledge about the way the training patterns fell in pattern space. A new training rule was introduced - the *delta rule* - which, it was claimed, could be generalised from the single unit/layer case to multilayer nets. This is now done heuristically for a two layer net of semilinear nodes*.



2 layer net as drawn last time

2 Backpropagation

2.1 Theory - where does it come from?

We analysed the delta rule with just one node. With more than one node on the output layer (N, say) the error has to be summed over all nodes

$$E_p = \frac{1}{2} \sum_{j=1}^{N} (t^j - y^j)^2 \tag{1}$$

*For a full proof, see the additional sheet given out in the lecture. This is *not* an examinable part of the course!

The idea is still to perform a gradient descent on the error considered as a function of the weights. This time, however, we are to take into account *all* the weights, for both *hidden* and *output* nodes. The former are nodes in the intermediate layer(s) which we do not have direct access to for the purposes of training (we can't say what their output will be). The output nodes are the ones which tell us the net's response to an input and to which we may show a supervisory or target value during training.

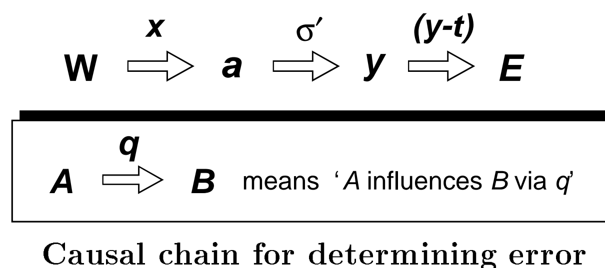
The analysis for the output nodes is just the same as in the delta rule given in eqn. (8) last time.

$$\Delta w_i^j = \alpha \sigma'(a^j)(t^j - y^j)x_i^j \tag{2}$$

Where a superscript has been introduced to denote which node is being described. This follows because the gradient of the error with respect a weight on the jth node can only be affected by that part of (1) which contains reference to that node.

In order to gain insight, it is useful to split the right hand side (RHS) of (2) up in the following way. The term $(t^j - y^j)$ represents a measure of the error on the jth node. The term $\sigma'(a^j)$ relates to how quickly (rate of change or slope) the activation can change the output (and hence the error). If this is small then we are on one of the 'tails' of the sigmoid and changing the activation won't change the output by much. If, however, it is large, then we can expect a rapid change for a given change in activation (see Qn. 3, problem sheet 3). The factor of x_i^j is related to, in turn, the amount that the ith input has affected the activation. If it is zero then that input cannot be 'held responsible' for the error and so the weight change should also be zero. If on the other hand, it is large (1, say) then the ith input had a large contribution to the activation which gave the error and so the weight needs to be changed by a correspondingly larger amount.

To summarise: x_i^j tells us how much the ith input was 'responsible for' the activation; $\sigma'(a^j)$ tells us, in turn, how fast the output is changing in response to changes in the activation and $(t^j - y^j)$ is the error on the jth node. It is therefore not unreasonable that the product of these gives us something that is a measure of the rate of change (slope) of the error with respect to the weight w_i^j. The situation is shown diagramatically below.



Causal chain for determining error

Using our previous notation we may combine two of these elements as follows [†]

$$\delta^j = \sigma'(a^j)(t^j - y^j) \tag{3}$$

The delta rule for output units may now be written

[†]Note the learning rate has now been excluded from the definition which we used when dealing with TLUs - this is, in fact, the normal convention.

$$\Delta w_i^j = \alpha \delta^j x_i \tag{4}$$

Consider now, the two layer net in the first diagram and, in particular, the kth hidden node. The problem in assigning a set of weight changes to this type of node is related to the so-called *credit assignment problem* - how much influence has this node had on the error. The resulting weight changes will be a result of including the right combination of 'responsibility' factors, rates of change and errors in the same way that these occurred for the output nodes. A full mathematical derivation is supplied in the supplement to these notes; this however, in itself, does not give insight. The purpose here is to shed some light on where the resulting formula comes from. As a start, we notice that, for the ith input to the hidden node, the value of the input will play a similar role as before so we might write

$$\Delta w_i^k = \alpha \delta^k x_i \tag{5}$$

and the task now is to find what goes into the factor δ^k

To this end, consider just a single output from the hidden node to an output node.

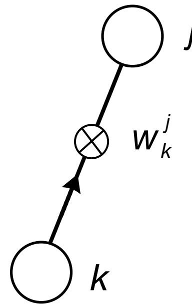


diagram of kth hidden and jth output nodes

The effect this node has on the error depends on two things: first how much it can influence the output of node j and, via this, how the output of node j affects the error. The more k can affect j, the greater the effect we expect there to be on the error, but this will only be significant if j is having some effect on the error at its output. The contribution that node j makes towards the error is, of course, expressed in the 'delta', for that node - δ^j. The influence that k has on j is given by the weight w_k^j. Therefore we may expect the find the product $w_k^j \delta^j$ in the expression for δ^k. However, the kth node may be giving output to several nodes and so the contributions to the error from all of these must be taken into account. Thus, we must sum these products over all j. Finally, the factor $\sigma'(a^k)$ will occur for exactly the same reasons that it did for the output nodes. This results in the following expression for δ^k

$$\delta^k = \sigma'(a^k) \sum_{j \in I_k} \delta^j w_k^j \tag{6}$$

where I_k is the set of nodes which take an input from the hidden node k. This set is called the *fan-out* of k. Using this in (5) gives us a means for calculating the weight changes for the hidden nodes.

2.2 Practice - using the training rules

It remains to develop a training algorithm around the rules we have developed. This basic iteration is the same as for the perceptron rule or delta rule

```
repeat
    for each training pattern
        train on that pattern
    end for loop
until the error is 'acceptably low'
```

Before examining the crucial step 'train on a pattern' a couple of points need comment. First, it is implied in the algorithm defined above that there is a fixed presentation sequence of training vectors. The alternative is to present vectors randomly. If we were to imagine our network in a real learning environment then this second option is more realistic. Empirically, however, it is often found that training is faster if the vectors are ordered in some way and that order is maintained in presentation. Second, what constitutes an acceptable error? One possible definition for Boolean training sets might be to ensure that all output nodes had responses in the correct one of the pair of intervals [0, 0.5], [0.5, 1] as defined by the target, since then, if we were to replace the sigmoid with a hard limiting threshold, the 'correct' response would be guaranteed. Another might simply prescribe some low value like 0.001. Whatever approach is used, one has to interpret the significance of the criterion.

The main step of training on a pattern may now be expanded into the following steps.

1. Present the pattern at the input layer
2. Let the hidden units evaluate their output using the pattern
3. Let the output units evaluate their output using the result in step 2) from the hidden units.

   The steps 1) - 3) are collectively known as the *forward pass* since information is flowing forward, in the natural sense, through the network.
4. Apply the target pattern to the output layer
5. Calculate the δ's on the output nodes according to (3)
6. Train each output node using gradient descent (4)
7. For each hidden node, calculate its δ according to (6)
8. For each hidden node, use the δ found in step 7) to train according to gradient descent (5).

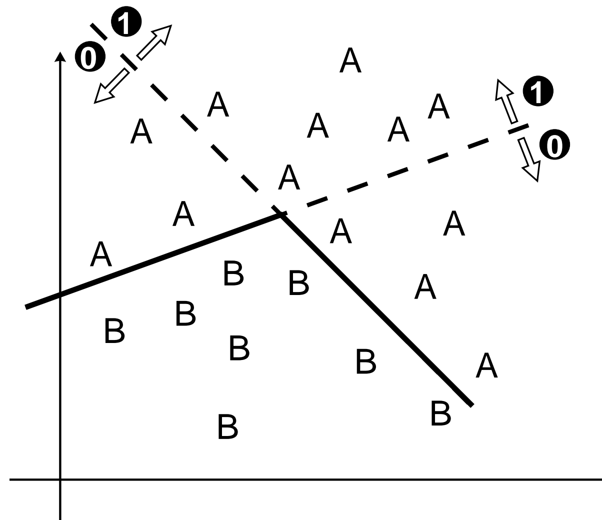
   Steps 4) - 8) are collectively known as the *backward pass*

Step 7) involves *propagating* the δ's from those output nodes in the hidden unit's fan-out *back* towards this node so that it can process them. This is where the name of the algorithm comes from.

Before going further it is useful to note some alternative terms used in the literature. The backpropagation (BP) algorithm is also known as *error backpropagation* or *back error propagation* or the *generalised delta rule*. The networks that get trained like this are sometimes known as *multilayer perceptrons* or MLPs.

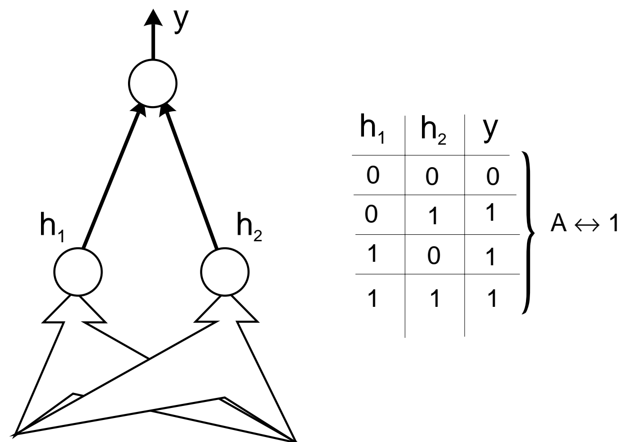
3 Non-linearly separable problems

The two layer net was originally introduced in the context of classifying more than 2 classes. Consider now, the following situation in pattern space



2 non linearly separable classes - 2 planes

The two classes A and B cannot be separated by a single hyperplane. In general we require an arbitrarily shaped *decision surface*. In the diagram, we have approximated this surface by two planes. We may now construct a two layer net to solve this problem. Each plane is determined by one of a pair of hidden nodes. Suppose each of these nodes learns to signal a '1' (or at least a value close to this with its sigmoid output) for class A. The output node can now classify A as a '1' if it learns the logical OR function.

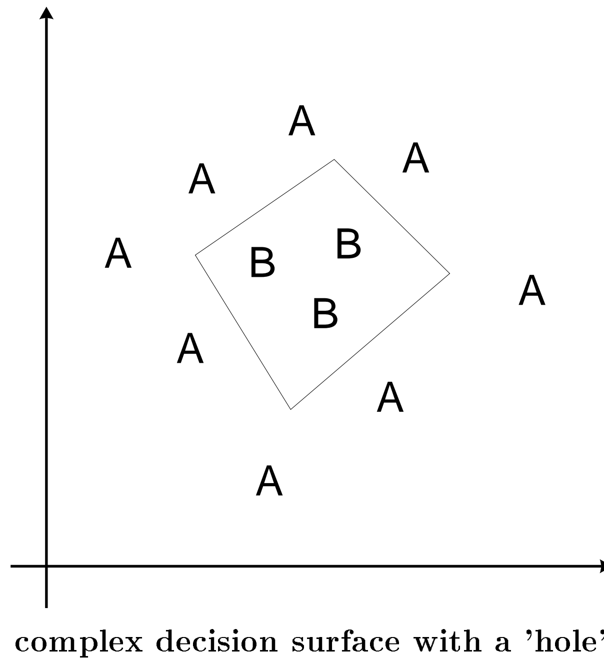


h_1	h_2	y
0	0	0
0	1	1
1	0	1
1	1	1

truth table for output node - OR gate

If the hidden units had coded class A in some other way then the output node would learn some other function in which a single corner of its pattern space square was 'lopped off'.

For more complex decision surfaces we need more hidden units



complex decision surface with a 'hole'

To summarise the power of the tools that have now been developed: we can train a multilayer net to perform categorisation of an arbitrary number of classes and with an arbitrary decision surface. All that is required is that we have a set of inputs and targets, that we fix the number of hyperplanes (hidden units) that are going to be used, and perform gradient descent on the error with the backpropagation algorithm. There are (as always) however, subtleties that emerge which make life difficult. The first of these concerns the number of hidden units used and relates to inadequate *training set generalisation*. The second concerns the nature of gradient descent.

4 Generalisation

Consider the situation in pattern space shown below

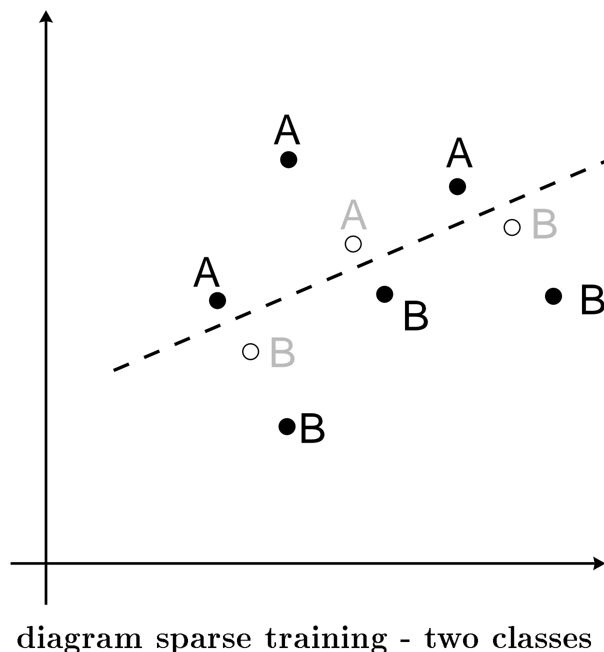
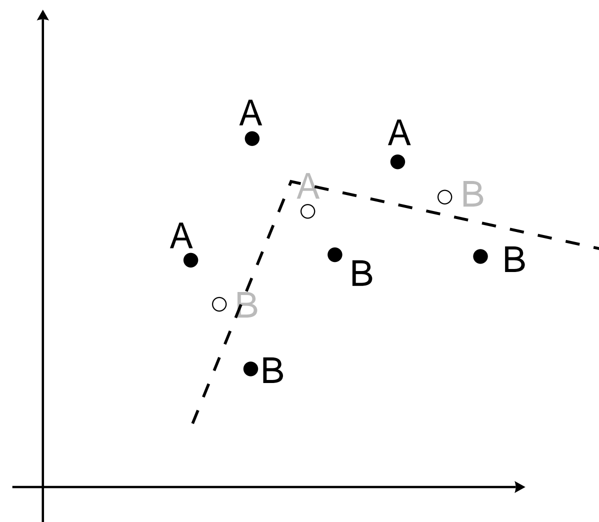


diagram sparse training - two classes

The training pattern are shown by solid dots and there are two classes A and B. Only one node is used to classify these. The circles in each class represent vectors which were not shown during training; these are *test patterns*. Representatives from each class of test data have been classified correctly, even though they were not seen during training. This is the power of the network approach and one of the main reasons for using it. The net is said to have *generalised* from the training data.

4.1 Overfitting the decision surface

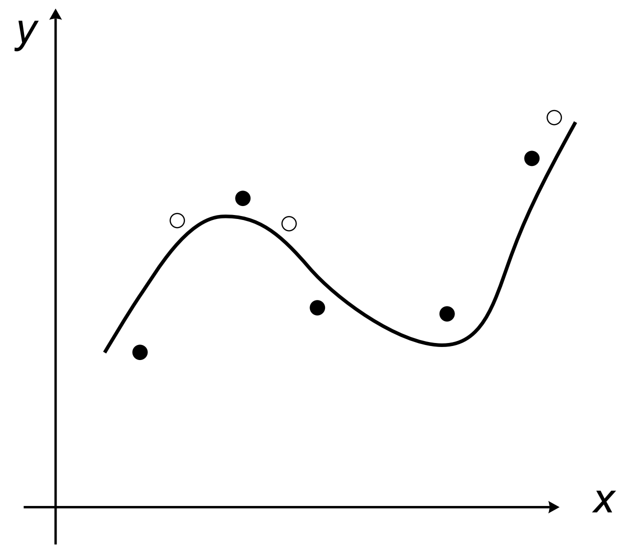
The correct classification of the test patterns shown in the last diagram would seem to vindicate the choice of a single hyperplane for the decision surface. Suppose, in fact, we had used two hyperplanes (two hidden units in a two layer net). In minimising the error, the planes might have aligned themselves as close to the training data as possible.



fitting two planes to A and B

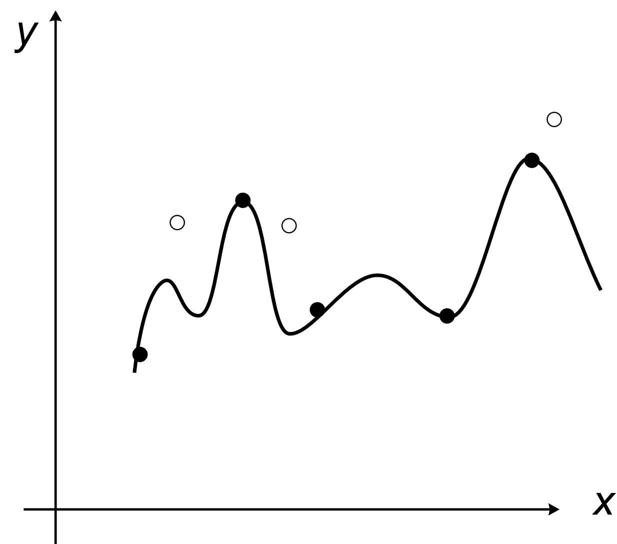
Some of the test data are now misclassified. The problem is that the network, with two hidden units, now has *too much* freedom and has fitted a decision surface to the training data which follows its intricacies in pattern space without extracting the underlying trends.

There is another way to view this in terms of the input-output function of the net. The diagram below shows, schematically, the output y of a binary (two-class) classifier, as a function of its input (this is a 1-D representation of the n-D input)



y against x for a binary classifier

The curve shown is the actual output in response to the input x, while the dots represent training data. If the curve had passed exactly through the training set, the error would be identically zero. Although this is not the case, the output has captured the underlying trend in the data. If we use more hidden units, the net has more freedom, its output can vary more quickly in response to a change in the input, and we might get a situation like that shown below



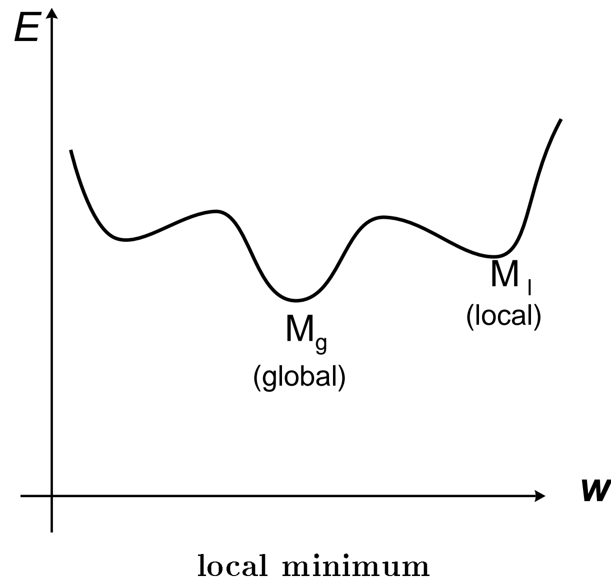
overfitting in x-y space

Now, although the curve fits almost exactly through the training data, giving almost zero error, the test data are poorly classified. The net has generalised poorly.

One of the questions that remained unanswered at the end of the last section was how to determine the number of hyperplanes or hidden units to use. At first, this might not have seemed a problem since it appears that we can always use more hidden units than are strictly necessary. There would simply be redundancy with more than one hidden node per hyperplane. This is the 'more of a good thing is better' approach. Unfortunately, as we have discovered here, things aren't so easy. Too many hidden nodes can make our net a very good look-up-table for the training set at the expense of any useful generalisation. How to fix the number of hidden nodes is an active research problem.

5 Local Minima

Consider the error function shown below.



local minimum

Suppose we start with a weight set for the network corresponding to point P. If we perform gradient descent, the minimum we encounter is the one at M_l, not that at M_g. M_l is called a *local minimum* and corresponds to a partial solution for the network in response to the training data. M_g is the *global minimum* we seek and, unless measures are taken to escape from M_l, M_g will never be reached. This problem will occur again in connection with feedback associative nets, where it will overcome by using noise in the training. Essentially, we opt for a situation where each move in weight space is governed, not only by the error gradient, but includes a random component so that sometimes we may go up the curve rather than down. In the type of nets being discussed here, however, the hope is that situations like that above don't occur.

6 Speeding up learning: the momentum term

The speed of learning is governed by the learning rate α. If this is increased too much, learning becomes unstable; the net oscillates back and forth across the error minimum. One way of overcoming the limitations thus imposed is to alter the training rule from 'pure' gradient descent to include a term which includes a proportion of the last weight change. The new rule is

$$\Delta w_i^j(n) = \alpha \delta^j x_i^j + \lambda \Delta w_i^j(n-1) \tag{7}$$

Thus, if the previous weight change $\Delta w_i^j(n-1)$ was large, so too will the new one $\Delta w_i^j(n)$. That is, the weight change carries along some momentum to the next iteration. This has a tendency to smooth out small fluctuations in the error-weight space (it is a low-pass filter). The parameter λ ('lambda') governs the contribution of the *momentum term.*

7 Further notes and reading

Backpropagation is probably the most well researched training algorithm in neural nets and forms the starting point for most people looking for a quick NN solution to a problem. There is therefore a wealth of literature on BP and its applications.

The theory and some toy applications are given in chapter 9 of PDP vol. 1.

The algorithm was actually discovered before Rumelhart & McClleland made it well-known, independently by P. Werbos and D. B. Parker. The references for these are a PhD thesis and an internal report at Stanford and were therefore not easily available.

One of the most powerful demonstrations of BP which helped it to fame was the NETtalk network of Sejnowski & Rosenberg which learned to translate written text to speech. (unfortunately this is a technical report and not easily available).

Another application is classification of sonar targets. Gorman, R.P. & Sejnowski, T. 'Analysis of hidden units in a layered network trained to classify sonar targets', Neural Networks, **1**, 75 – 89. (I have this).

5: Associative memories - the Hopfield net

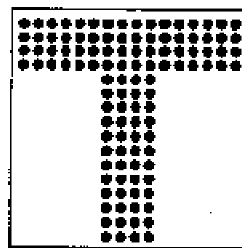
Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

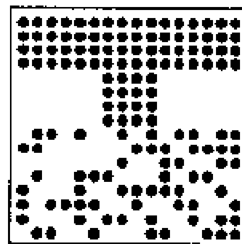
1 The nature of associative memory

'Remembering' something in common parlance usually consists of associating something with a sensory cue. For example, someone may say something, like the name of a celebrity, and we immediately recall a chain of events or some experience related to the celebrity - we may have seen them on TV recently for example. Or, we may see a picture of a place visited in our childhood and the image recalls memories of the time. The sense of smell (olfaction) is known to be especially evocative in this way.

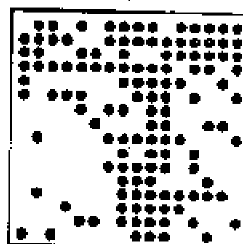
On a more mundane level, but still in the same category, we may be presented with a partially obliterated letter, or one seen through a window when it is raining (letter + noise) and go on to recognise the letter.



Original 'T'



half of image
corrupted by
noise



20% corrupted
by noise
(whole image)

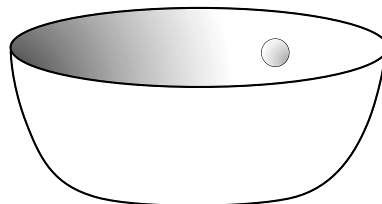
slide of 'T's

The common paradigm here may be described as follows. There is some underlying collection of data which is ordered and interrelated in some way and which is stored in memory. The data may be thought of, therefore, as forming a stored pattern. In the recollection examples above, it is the cluster of memories associated with the celebrity or the phase in childhood. In the case of character recognition, it is the parts of the letter (pixels) whose arrangement is determined by an archetypal version of the letter. When part of the pattern of data is presented in the form of a sensory cue, the rest of the pattern (memory) is recalled or *associated* with it. Notice that it often doesn't matter which part of the pattern is used as the cue, the whole pattern is always restored.

Conventional computers (von Neumann machines) can perform this function in a very limited way. The typical software for this is usually referred to as a database. Here, the 'sensory cue' is called the *key* which is to be searched on. For example, the library catalogue is a database which stores the authors, titles, classmarks and data of publication of books and journals. We may search on any one of these discrete items for a catalogue entry by typing the complete item after selecting the correct option from a menu. Suppose now we have only the fragment 'ion, Mar' from the full title 'Vision, Marr D.'. There is no way that the database can use this fragment of information to even start searching. We don't know if it pertains to the author or the title, and even if we did, we might get titles or authors that start with 'ion'. The kind of input to the database has to be very specific and complete.

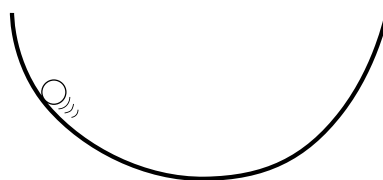
2 A physical analogy with memory

The networks that are used to perform associative recall are specific examples of a wider class of physical systems which may be thought of as doing the same thing. This allows the net operation to be viewed as a the dynamics of a physical system and its behaviour to be described in terms of the network's 'energy'. Consider a bowl in which a ball bearing is allowed to roll freely



bowl and ball bearing in 3D

This is more easily drawn using a 2D cross section



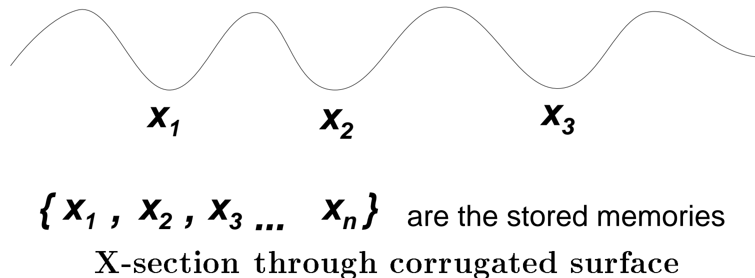
2d X-section of bowl

Suppose we let the ball go from a point somewhere up the side of the bowl with, possibly, a push to one side as well. The ball will roll back and forth and around the bowl until it comes to rest at the bottom.

The physical description of what has happened may be couched in terms of the energy of the system. The ball initially has some *potential* energy. That is work was done to push it up the side of the bowl to get it there and it now has the potential to gain speed and acquire energy. When the ball is released, the potential energy is released and the ball rolls around the bowl (it gains *kinetic* energy). Eventually the ball comes to rest where its energy (potential and kinetic) is zero. (The kinetic energy gets converted to heat via friction with the bowl and the air). The main point is that the ball comes to rest in the same place every time and this place is determined by the energy minimum of the system (ball + bowl). The resting state is said to be *stable* because the system remains there after it has been reached

There is another way of thinking of this process which ties in with our ideas about memory. We suppose that the ball comes to rest in the same place each time because it 'remembers' where the bottom of the bowl is. We may push the analogy further by giving the ball a coordinate description. Thus, its position or *state* at any time is given by the three coordinates (x, y, z) or the position vector $\mathbf{x}$. The location of the bottom of the bowl, $\mathbf{x}_0$ represents the pattern which is stored. By writing the ball's vector as the sum of $\mathbf{x}_0$ and a displacement $\Delta\mathbf{x}$ thus, $\mathbf{x} = \mathbf{x}_0 + \Delta\mathbf{x}$, we may think of the ball's initial position as representing the partial knowledge or cue for recall, since it approximates to the memory $\mathbf{x}_0$.

If we now use a corrugated surface instead of a single depression (the bowl) we may store many 'memories'.



X-section through corrugated surface

If now the ball is started somewhere on this surface, it will eventually come to rest at the local depression which is closest to its initial starting point. That is it evokes the stored pattern which is closest to its initial partial pattern or cue. Once again, this is an energy minimum for the system.

There are therefore two complementary ways of looking at what is happening. One is to say that the system falls into an energy minimum; the other is that it stores a set of patterns and recalls that which is closest to its initial state. If we are to build a network which behaves like this we must include the following key elements

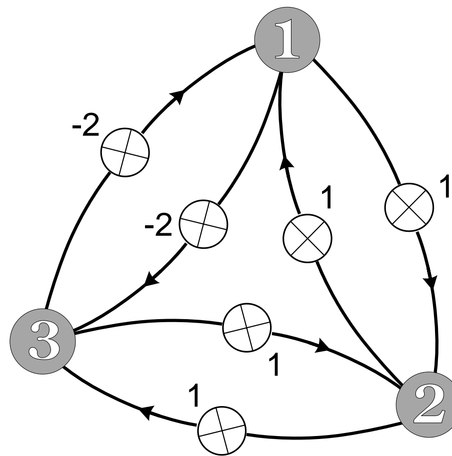
1. It is completely described by a *state vector* $\mathbf{v} = (v_1, v_2, \ldots, v_n)$

2. There are a set of *stable states* $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_1, \ldots, \mathbf{v}_n$ These will correspond to the stored patterns and, in, the corrugated surface example, were the bottoms of the depressions in the surface.

3. The system evolves in time from any arbitrary starting state $\mathbf{v}$ to one of the stable states, and this may be described as the system decreasing its energy E. This corresponds to the process of memory recall.

3 The Hopfield net

Consider the network consisting of three TLU nodes shown below



3 node Hopfield net

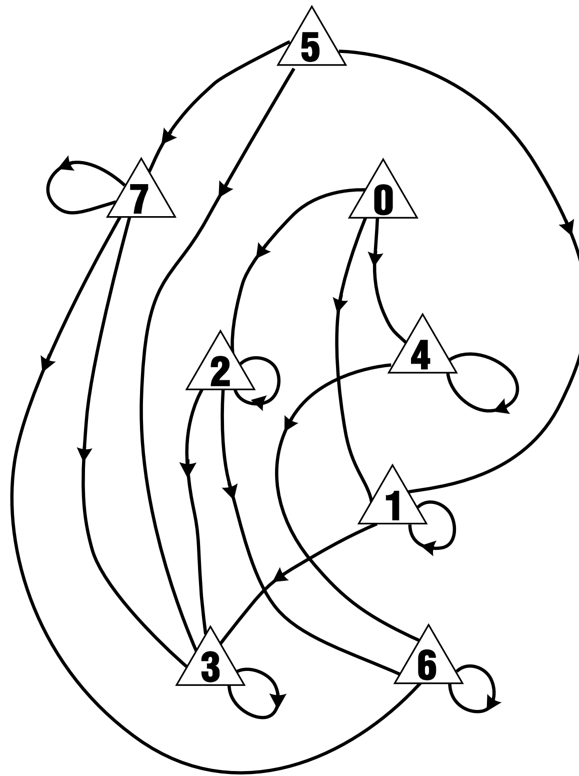
Every node is connected to every other node (but *not* to itself) and the connection strengths or weights are symmetric in that the weight from node i to node j is the same as that from node j to node i. That is, $w_{ij} = w_{ji}$, and $w_{ii} = 0$ for all i, j.* Notice that the flow of information in this net is not in a single direction as it has been in the nets dealt with so far. It is possible for information to flow from a node back to itself via other nodes. That is, there is feedback in the network and so they are known as feedback or *recurrent* nets as opposed to *feedforward* nets which were the subject of the Backpropagation algorithm.

The state of the net at any time is given by the vector of the node outputs (x_1, x_2, x_3). Suppose we now start this net in some initial state and choose a node at random and let it update its output or 'fire'. That is, it evaluates its activation in the normal way and outputs a '1' if this is greater than or equal to zero and a '0' otherwise. The net now finds itself either in the same state as it started in, or in a new state which is at Hamming distance one from the old. We now choose a new node at random to fire and continue in this way over many steps. What will the behaviour of the net be? For each state, we may evaluate the next state given each of the three nodes fires. This gives the following table.

State				**New state** (after node has fired)		
Number	vector					
	x_1	x_2	x_3	Node 1	Node 2	Node 3
0	0	0	0	4	2	1
1	0	0	1	1	3	1
2	0	1	0	6	2	3
3	0	1	1	3	3	3
4	1	0	0	4	6	4
5	1	0	1	1	7	3
6	1	1	0	6	6	6
7	1	1	1	3	7	6

*The weight from node i to node j is sometimes also denoted by w_i^j

This information may be represented in graphical form as a *state transition diagram.*



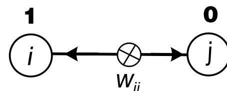
state transition diagram for 3 node net

States are represented by the circles with their associated state number. Directed arcs represent possible transitions between states and the number alongside each arc is the probability that each transition will take place. The states have been arranged in such a way that transitions tend to take place down the diagram; this will be shown to reflect the way the system decreases its energy. The important thing to notice at this stage is that, no matter where we start in the diagram, the net will eventually find itself in one of the states '3' or '6'. These reenter themselves with probability 1. That is they are stable states - once the net finds itself in one of these it stays there. The state vectors for '3' and '6' are (0,1,1) and (1,1,0) respectively and so these are the 'memories' stored by the net.

3.1 Defining an energy for the net

The dynamics of the net are described perfectly by the state transition table or diagram. However, greater insight may be derived if we can express this in terms of an energy function and, using this formulation, it is possible to show that stable states will always be reached in such a net.

Consider two nodes i, j in the net which are connected by a positive weight and where j is currently outputting a '0' while i is outputting a '1'.



two nodes in conflict

If j were given the chance to update or fire, the contribution to its activation from i is positive and this may well serve to bring j's activation above threshold and make it output a '1'. A similar situation would prevail if the initial output states of the two nodes had been reversed since the connection is symmetric. If, on the other hand, both units are 'on' they are reinforcing each other's current output. The weight may therefore be thought of as fixing a constraint between i and j that tends to make them both take on the value '1'. A negative weight would tend to enforce opposite outputs. One way of viewing these networks is therefore as *constraint satisfaction* nets.

This idea may be captured quantitatively in the form of a suitable energy function. Define

$$e_{ij} = -w_{ij}x_i x_j \tag{1}$$

The values that e_{ij} take are given in the table below

x_i	x_j	e_{ij}
0	0	0
0	1	0
1	0	0
1	1	$-w_{ij}$

If the weight is positive then the last entry is negative and is the lowest value in the table. If e_{ij} is regarded as the 'energy' of the pair ij then the lowest energy occurs when both units are on which is consistent with the arguments above. If the weight is negative, the '11' state is the highest energy state and is not favoured. The energy of the net is found by summing over all pairs of nodes

$$E = \sum_{\text{pairs}} e_{ij} = -\sum_{\text{pairs}} w_{ij}x_i x_j \tag{2}$$

This may be written

$$E = -\frac{1}{2}\sum_{i,j} w_{ij}x_i x_j \tag{3}$$

Since the sum includes each pair twice (as $w_{ij}x_i x_j$ and $w_{ji}x_j x_i$) and $w_{ij} = w_{ji}$, $w_{ii} = 0$.

It is now instructive to see what the change in energy is when a node fires. Suppose node k is chosen to be updated. Write the energy E by singling out the terms involving this node.

$$E = -\frac{1}{2}\sum_{\substack{i\neq k\\ j\neq k}} w_{ij}x_i x_j - \frac{1}{2}\sum_i w_{ki}x_k x_i - \frac{1}{2}\sum_i w_{ik}x_i x_k \tag{4}$$

Now, because $w_{ik} = w_{ki}$, the last two sums may be combined

$$E = -\frac{1}{2}\sum_{\substack{i\neq k\\ j\neq k}} w_{ij}x_i x_j - \sum_i w_{ki}x_k x_i \tag{5}$$

For ease of notation, denote the first sum by S and take the x_k outside the sum since it is constant throughout, then

$$E = S - x_k \sum_i w_{ki} x_i \tag{6}$$

but the sum here is just the activation of the kth node so that

$$E = S - x_k a^k \tag{7}$$

Let the energy after k has updated be E' and the new output be x'_k. Then

$$E' = S - x'_k a^k \tag{8}$$

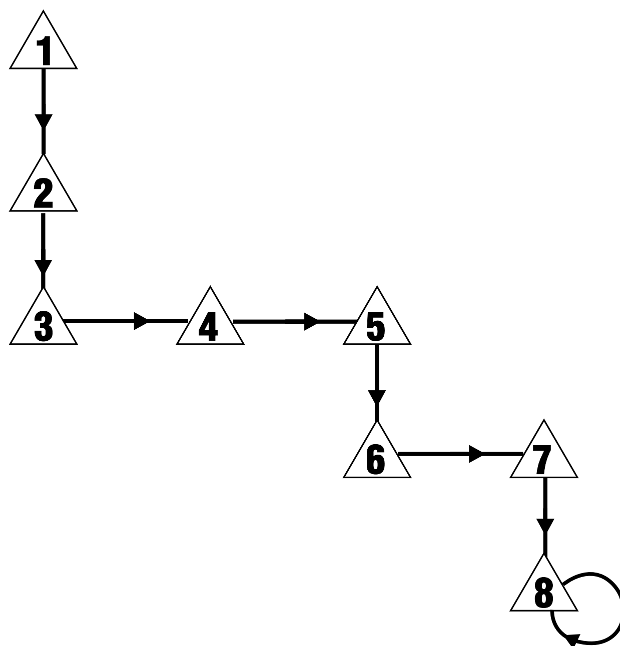
Denote the change in energy $E' - E$ by ΔE and the change in output $x'_k - x_k$ by Δx_k, then subtracting (7) from (8)

$$\Delta E = -\Delta x_k a^k \tag{9}$$

There are now two cases to consider

1. $a^k \geq 0$. Then the output goes from '0' to '1' or stays at '1'. In either case $\Delta x_k \geq 0$. Therefore $\Delta x_k a^k \geq 0$ and so, $\Delta E \leq 0$
2. $a^k < 0$. Then the output goes from '1' to '0' or stays at '0'. In either case $\Delta x_k \leq 0$. Therefore, once again $\Delta x_k a^k \geq 0$ and $\Delta E \leq 0$

Thus, for any node being updated we always have $\Delta E \leq 0$ and so the energy of the net decreases or stays the same. But the energy is bounded below by a value obtained by putting all the $x_i, x_j = 1$ in (3). Thus E must reach some fixed value and then stay the same. Once this has occurred, it is possible for further changes in the network's state to take place since $\Delta E = 0$ is still allowed. However, for this to be the case ($\Delta x_k \neq 0$ and $\Delta E = 0$) we must have $a^k = 0$. This implies the change must be of the form $0 \to 1$. There can be at most N (of course there may be none) of these changes, where N is the number of nodes in the net. After this there can be no more change to the net's state and a stable state has been reached.

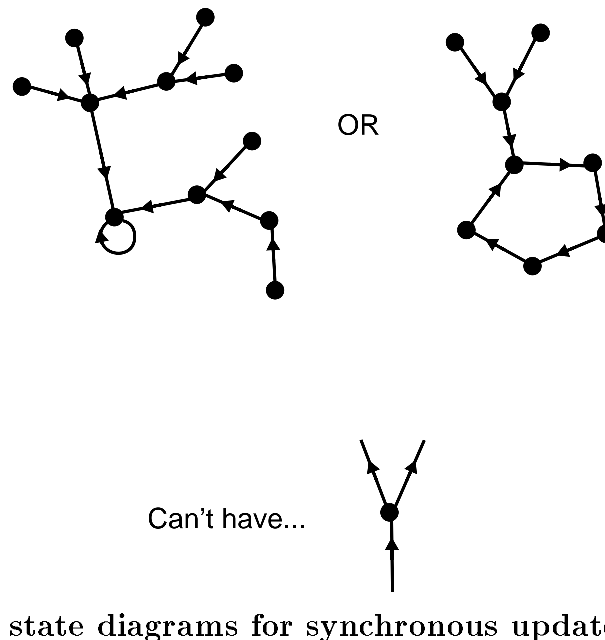


state transitions to stable state

In the example given above, all state have zero energy except for state 5 which has energy 2, and the stable states 3 and 6 which have energy -1.

3.2 Asynchronous vs. synchronous update

So far we have allowed only a single node to update or fire at any time step. All nodes are possible candidates for update and so they operate *asynchronously*; that is, there is no coordination between them in time. The other extreme case occurs if we make all nodes fire at the same time, in which case we say there is *synchronous* update. To do this, we must ensure that each nodes previous output is available to the rest of the net until all nodes have evaluated their activation and been updated. It is therefore necessary to store both the current state vector *and* the next state vector. The behaviour is now deterministic; given any state, a state transition occurs to a well defined next state, there being no probabilistic behaviour. The analysis, in terms of energy changes at each update, given above no longer applies but there is now a simplified type of state diagram in which only a single arc emerges from any state. This allows us to predict, once again, the general type of behaviour. In particular, state-cycles occur again but now there is the possibility for *multiple-state cycles.*



state diagrams for synchronous update

These may be useful in storing sequences of events or patterns. A little thought will show that the (single) state cycles remain the same under synchronous dynamics so the single stored patterns remain the same under both dynamics.

3.3 Ways of inputting information

So far it has been assumed that the net is started in some initial state (the memory cue) and the whole net allowed to run freely until a state cycle is encountered (recall slide of noisy 'T'). There is another possibility in which some part of the net has its outputs fixed while the remainder is allowed to update. The part that is fixed is said to be *clamped* and if the

clamp forms part of a state cycle, the remainder (unclamped) part of the net will complete the pattern stored at that cycle (recall slide of partially correct 'T'). Which mode is used will depend on any prior knowledge about parts of the cue being uncorrupted or noise free.

The problem of how to fix the weights in Hopfield nets will be dealt with next time.

References

Aleksander, I. and Morton, H. (1990). *Neural Computing.* Chapman hall.
Quite good on Hopfield nets and contains a similar example to the one given in these notes.

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational properties. *Proceedings of the National Academy of Sciences of the USA*, 79:2554 – 2588. Hopfield has another, related, model which uses continuous outputs. Beware when reading the literature which model is being discussed.

McEliece, R., Posner, E., Rodemich, E., and Venkatesh, S. (1987). The capacity of the hopfield associative memory. *IEEE Transactions on Information Theory*, IT-33:461 – 482.
There are many papers on this area, but this has some non- technical material near the beginning before getting into the welter of maths needed for this kind of analysis.

6: Hopfield nets (contd.)

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

1 Finding the weights

So far we have described the dynamics of Hopfield nets but nothing has been said about the way the weights are established for a particular problem. In his original paper, Hopfield (1982) did not give a method for training the nets, rather he gave a *prescription* for making a weight set, given a set of patterns to be stored. Here, we shall relate the storage prescription, later on, to a biologically inspired learning rule - the Hebb rule - and show that the nodes may also be trained individually using the delta rule.

1.1 The storage prescription

The rationale behind the prescription is based on the desire to capture, in the value of the weights, local correlations between node outputs when the net is in one of the required stable states. Recall that these correlations also gave rise to the energy description and the same kind of arguments will be used again.

Consider two nodes which, on average over the required pattern set, tend to take on the same value. That is, they tend to form either the pair (0, 0) or (1, 1). The latter pairing will be reinforced by there being a positive weight between the nodes, since each one is then making a positive contribution to the others activation which will tend to foster the production of a '1' at the output. Now suppose that the two nodes, on average, tend to take on opposite values. That is they tend to form either the pair (0, 1) or (1, 0). Both pairings are reinforced by a negative weight between the two nodes, since there is a negative contribution to the activation of the node which is 'off' from the node which is 'on', supporting the former's output state of '0'. Note that, although the pairing (0, 0) is not actively supported by a positive weight *per se*, a negative weight would support the mixed output pair-type just discussed.

These observations may be encapsulated mathematically in the following way. First we introduce an alternative way of representing binary quantities. Normally these have been denoted by 0 or 1. In the *polarised* or *spin*[*] representation they are denoted by -1 and 1 respectively, so there is the correspondence $0 \leftrightarrow -1, 1 \leftrightarrow 1$. Now let v_i^p, v_j^p be components

[*]This name is derived from the fact that Hopfield nets have many similarities with so-called *spin glasses* in physics, the prototypical example of which is a collection of magnetic domains whose polarisation of ±1 is determined by the average spin of the electrons in each domain

of the pth pattern to be stored where these are in the spin representation. Consider what happens if the weight between the nodes i and j is given by

$$w_ij = \sum_p v_i^p v_j^p \tag{1}$$

Where the sum is over all patterns p to be stored. If, on average, the two components take on the same value then the weight will be positive since we get terms like 1×1 and -1×-1 predominating. If, on the other hand, the two components, on average, take on opposite values we get terms like -1×1 and 1×-1 predominating which gives a negative weight. This is just what was required according to the arguments given above. Equation (1) is therefore the storage prescription used with Hopfield nets. Note that, the same weights would accrue if we had tried to learn the inverse of the patterns formed by taking each component of every pattern and changing it to the opposite value. The net therefore, always learns the patterns *and* their inverses.

1.2 The Hebb rule

The use of (1) which is an algorithm or recipe for fixing the weights without adapting to a training set may appear to run counter to the ideas being promoted in the connectionist cause. It is possible, however, to view the prescription in (1) as a short-cut to a process of adaptation which would take place if we were to obey the following training algorithm

1. present the components of one of the patterns to be stored at the outputs of the corresponding nodes of the net.

2. If two nodes have the same value then make a small positive increment to the internode weight. If they have opposite values then make a small negative decrement to the weight.

Steps 1) and 2) then get repeated many times with a different pattern selection in 1). Symbolically step 2) (which is the learning rule) may be written

$$\Delta w_{ij} = \alpha v_i^p v_j^p \tag{2}$$

where, as usual α is a rate constant and $0 < \alpha < 1$. It is clear that the storage prescription is just the result of adding all the weight changes that would accumulate under this scheme if enough pattern presentations were made. The rule in (2) is one of a family of rules known as *Hebb rules* after D. O. Hebb. The characteristic of such a rule is that it involves the product of a pair of node activations or outputs.

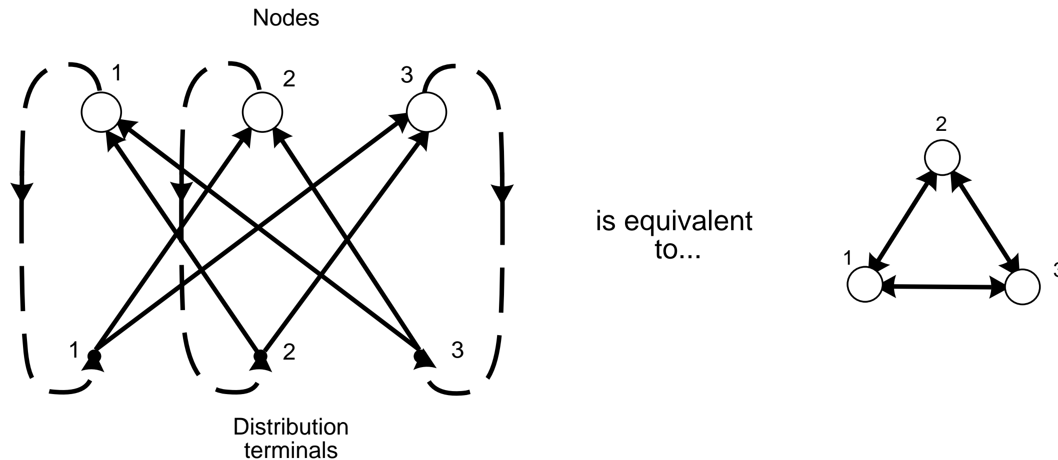
As a variation, suppose we had used the usual Boolean representation for the components x_i^p so that x_i^p is 0 or 1 The Hebb rule would now be $\Delta w_{ij} = \alpha x_i^p x_j^p$. Interpreting this, we have that the change in weight is only ever positive and only occurs if both nodes are firing (output '1'). This is, in fact closer to the original rule proposed by Hebb (1949) in a neurophysiological context. In his book *The Organization of behaviour* Hebb postulated that

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

That is, the correlation of activity between two cells is reinforced by increasing the synaptic strength (weight) between them. Simpson (course book) contains a list of Hebb rule variants.

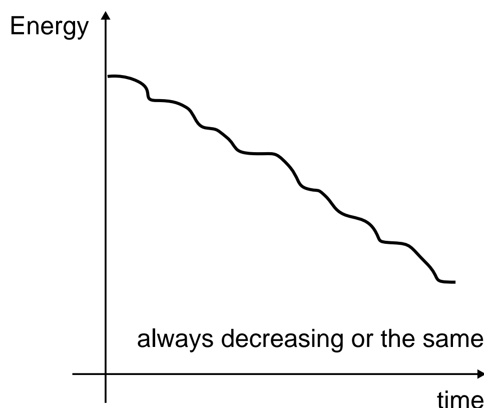
1.3 Using the delta rule

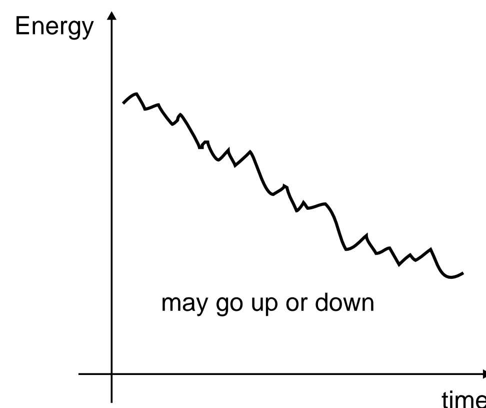
We may draw the connections in, say, a 3 node Hopfield net as follows.



3 node Hopfield net as feedforward with recurrence

Each node may now be thought of as taking part in some input- output function between the distribution terminals and the node outputs. They may therefore each be trained with the delta rule. If the corresponding training set for each one is linearly separable then the set of stable states may be learnt. However, there is no guarantee now that $w_{ij} = w_{ji}$. The change in energy at each node update is now no longer necessarily less than or equal to zero. The consequence is that, given the stable states have been trained, the system moves through state space with decreasing error towards a stable state but has, superimposed on this, some noise.





Symmetric

Asymmetric

energy v time for symmetric and asymmetric nets

2 Storage capacity

How good is the storage prescription (1) at storing the patterns so that they are stable states? Clearly, as the number of patterns m increases, the chances of accurate storage must decrease. In some empirical work in his 1982 paper, Hopfield showed that about half the memories were stored accurately in a net of N nodes if $m = 0.15N$. The other patterns did not get stored as stable states. In a more rigorous piece of analysis McCliece et al. (1987) showed theoretically that, if we require almost all the required memories to be stored accurately, then the maximum number of patterns m is $N/2 \log N$. For $N = 100$ this gives $m = 11$.

Suppose a pattern which was required to be stored did not, in fact produce a stable state and we start the net in this state. The net must evolve to *some* stable state and this is usually not related to any of the original patterns used in the prescription. The stable state represents a spurious energy minimum of the system - one that is not there by design.

3 The analogue Hopfield model

In a second important paper (Hopfield, 1984) Hopfield introduced a variant of the discrete time model discussed so far which uses nodes described by their rate of change of activation. This kind of node was discussed in the last part of lecture 1 but we review it here. Denote the sum of excitation from other nodes for the j node by s^j so that

$$s^j = \sum_i w_{ji} x_i \tag{3}$$

then the rate of change of activation da^j/dt is given by

$$\frac{da^j}{dt} = ks^j - ca^j \tag{4}$$

here, k and c are constant. The first term (if positive) will tend to make the activation increase while the second term is a decay term (see lecture 1). The output y^j is then just the sigmoid of the activation as usual. Hopfield also introduced the possibility of external input at this stage and a variable threshold.

In the previous TLU model, the possible states of an N node net are just the corners of the N-dimensional hypercube. In the new model, because the outputs can take any values between 0 and 1, the possible states include now, the interior of the hypercube. Hopfield defined an energy function for the new network and showed that if the inputs and thresholds were set to zero, as in the TLU discrete time model, and if the sigmoid was quite 'steep', then the energy minima were confined to regions close to the corners of the hypercube and these corresponded to the energy minima of the old model.

There, however, are two advantages of the new model. The first, is that the use of the sigmoid and time integration make more contact possible with real biological neurons. The second is that it is possible to build the new neurons out of simple, readily available hardware. In fact, Hopfield writes the equation for the dynamics - eqn (4) - as if it were built from an operational amplifier and resistor network. This kind of circuit was the basis of several implementations - see for example Graf et al. (1987).

4 Combinatorial optimisation

Another innovation made by Hopfield was to show how to solve large combinatorial optimisation problems on neural nets (Hopfield and Tank, 1985). The classical example of this is the so- called Travelling salesman Problem (TSP). Here, a travelling salesman has to visit each of a set of cities in turn in such a way as to minimise the total distance travelled. Each city must be visited once and only once. This kind of problem is computationally difficult in a technical sense (NP-complete) in that the time to solution scales with the number cities faster than the time t raised to any fixed power and therefore might scale like e^t.

The solution of the TSP consists of a sequence of cities. The problem for N cities may be coded into an N by N network as follows. Each row of the net corresponds to a city. The position of the city in the solution sequence is given by putting a '1' at the corresponding place in the row and 0's everywhere else in that row. Thus, if the city corresponding to row 5 was 7th in the sequence there would be a 1 in the 7th place of the 5th row and zeros everywhere else. Note that most states of the net do not correspond to valid tours - there must be only one '1' per row. The problem then, is to construct an energy function (and hence a set of weights) which lead to stable states (states of lowest energy) of the network that, not only express valid city tours, but which also are tours of short length. The validity criterian results in negative weights (inhibition) between nodes in the same row, and between nodes in the same column. The path-length criterion leads to inhibition between adjacent columns (cities in a path) proportional to the path length between the cities (row positions). The net is now allowed to run until an energy minimum is reached which should now correspond to a solution of the problem.

References

Graf, H., Hubbard, W., Jackel, L., and deVegvar P.G.N (1987). A cmos associative memory chip. In *1st Int. Conference Neural Nets, San Diego.*
(I have this).

Hebb, D. (1949). *The Organization of behaviour.* John Wiley.

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational properties. *Proceedings of the National Academy of Sciences of the USA*, 79:2554 – 2588. Hopfield has another, related, model which uses continuous outputs. Beware when reading the literature which model is being discussed.

Hopfield, J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences of the USA*, 81:3088 – 3092.

Hopfield, J. and Tank, D. (1985). Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141 – 152.

McEliece, R., Posner, E., Rodemich, E., and Venkatesh, S. (1987). The capacity of the hopfield associative memory. *IEEE Transactions on Information Theory*, IT-33:461 – 482.
There are many papers on this area, but this has some non- technical material near the beginning before getting into the welter of maths needed for this kind of analysis.

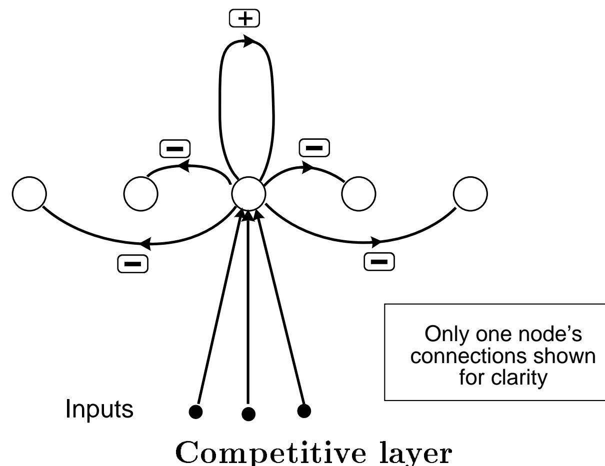
7: Competition and self-organisation: Kohonen nets

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

1 Competitive dynamics

Consider a layer or group of units as shown in the diagram below.



Competitive layer

Each cell receives the same set of inputs from an input layer and there are intralayer or *lateral* connections such that each node is connected to itself via an excitatory (positive) weight and inhibits all other nodes in the layer with negative weights.

Now suppose a vector $\mathbf{x}$ is presented at the input. Each unit now computes a weighted sum s of the inputs provided by this vector. That is

$$s = \sum_i w_i x_i \tag{1}$$

In vector notation this is, of course, just the dot product $\mathbf{w} \cdot \mathbf{x}$. This is the way of looking at things which will turn out to be most useful. Then some node k, say, will have a value of s larger than any other in the layer. It is now claimed that, if the node activation is allowed to evolve by making use of the lateral connections, then node k will develop a maximal value for a while the others get reduced. The time evolution of the node is usually governed by an equation which determines the rate of change of the activation (Lecture 1, section 'Introducing time'). This must include the input from the lateral connections as well

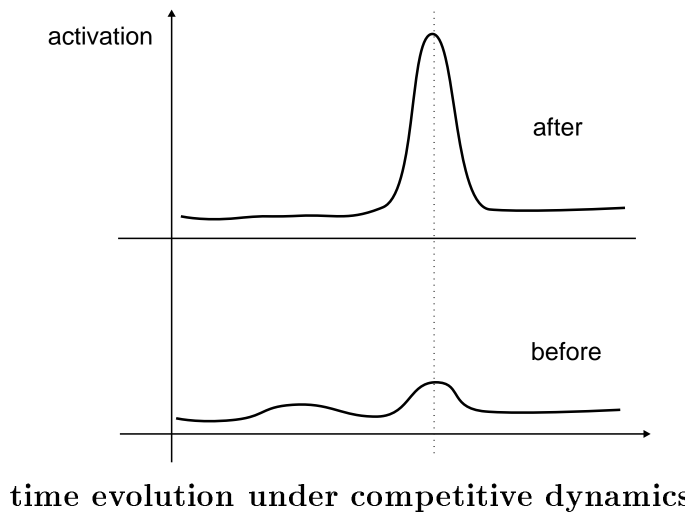
as the 'external' input given by s. Thus if l is the weighted sum of inputs from the lateral connections

$$\frac{da}{dt} = \beta_s s + \beta_l l - \gamma a \tag{2}$$

Recall that da/dt is the rate of change of a. There will usually be a sigmoid output relation $y = \sigma(a)$

What happens is that the node with greatest excitation s from the input has its activation increased directly by this and indirectly via the self-excitatory connection. This then inhibits the neighbouring nodes, whose inhibition of k is then further reduced. This process is continued until a stability is reached. There is therefore a 'competition' for activation across the layer and the network is said to evolve via *competitive dynamics.* Under suitable conditions, the nodes whose input s was less than that on the 'winning node' k will have their activity reduced to zero. The net is then sometimes referred to as 'winner-takes-all' net, since the node with largest input 'wins' all the available activity.

If the net's activity is represented in profile along the string of nodes then an initial situation in part a) of the diagram below will evolve into the situation shown in part b).



time evolution under competitive dynamics

Competitive dynamics are obviously useful in enhancing the activation 'contrast' over a network layer and singling out the node which is responding most strongly to its input. We now examine how this process may be useful in a learning situation.

2 Competitive learning

Consider a training vector set whose vectors all have the same length, and suppose, without loss of generality, that this is one. Recall that the length $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ is given by

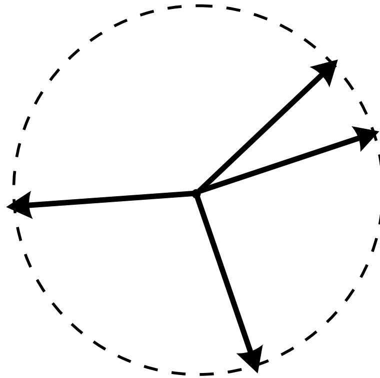
$$\|\mathbf{x}\| = \sum_i x_i^2 \tag{3}$$

A vector set for which $\|\mathbf{x}\| = 1$ for all $\mathbf{x}$ is said to be *normalised.* If the components are all positive or zero * then this is approximately equivalent to the condition

*this is consistent with the interpretation of the input as derived from the output of a previous layer

$$\sum_i x_i = 1 \tag{4}$$

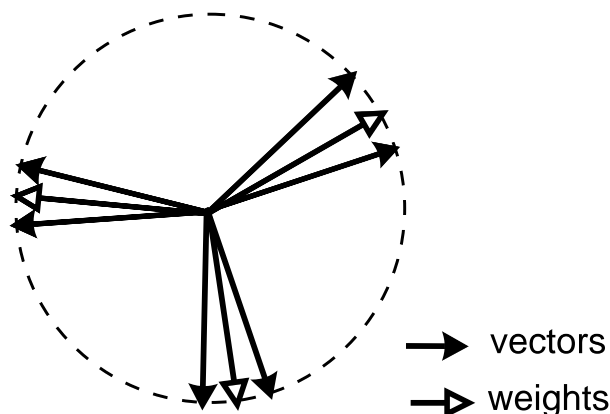
Since the vectors all have unit length, they may be represented by arrows from the origin to the surface of the unit (hyper)sphere.



vectors on unit hypersphere

Suppose now that a competitive layer has had its weight vectors normalised according to (4). Then these vectors may also be represented on the same sphere.

What is required for the net to encode the training set is that the weight vectors become aligned with any clusters present in this set and that each cluster is represented by at least one node. Then, when a vector is presented to the net there will be a node, or group of nodes, which respond maximally to the input and which respond in this way only when this vector is shown at the input.



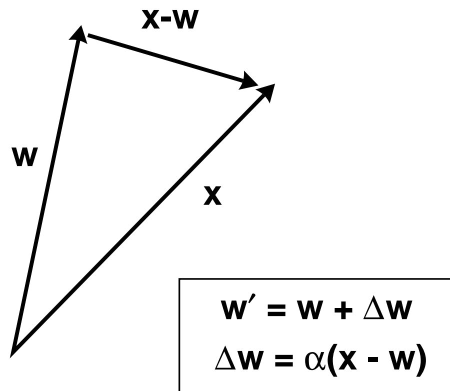
weights and vectors aligned

If the net can learn a weight vector configuration like this, without being told explicitly of the existence of clusters at the input, then it is said to undergo a process of *self-organised* or *unsupervised* learning. This is to be contrasted with nets which were trained with the delta rule or BP where a target vector or output had to be supplied.

In order to achieve this goal, the weight vectors must be rotated around the sphere so that they line up with the training set. The first thing to notice is that this may be achieved in an gradual and efficient way by moving the weight vector which is closest (in an angular sense) to the current input vector towards that vector slightly. The node k with the closest vector is that which gives the greatest input excitation s since this is just the dot product of

the weight and input vectors. As shown below, the weight vector of node k may be aligned more closely with the input if a change $\Delta\mathbf{w}$ is made according to

$$\Delta\mathbf{w} = \alpha(\mathbf{x} - \mathbf{w}) \tag{5}$$



vector triangle - weights and inputs

Now it would be possible to use a supervisory computer to decide which node had the greatest excitation s but it is more satisfactory if the net can do this itself. This is where the competitive dynamics comes in to play. Suppose the net is winner- take-all so that the winning node has value 1 and all the others have value close to zero. After letting the net reach equilibrium under the lateral connection dynamics we now enforce the rule

$$\Delta\mathbf{w} = \alpha(\mathbf{x} - \mathbf{w})y \tag{6}$$

across the whole net. Then there will only be a single node (the one whose dot-product s was greatest) which has $y = 1$ and for which the weight change is the same as in (6). All other nodes will have $y = 0$ and so there weight change will also be zero. The stages in learning (for a single vector presentation) are then

1. apply vector at input to net and evaluate s for each node.

2. update the net (in practice, in discrete steps) according to (2)for a finite time or until it reaches equilibrium.

3. train all nodes according to (6)

There are a few points about the learning rule worth noting. First, if the weights are initially normalised according to (4) and the input vectors are normalised in the same way, then the normalisation of the weights is preserved under the learning rule. The change in the length of $\mathbf{w}$ is given by the sum of the changes in its components

$$\sum_i \Delta w_i = \alpha y \left(\sum_i x_i - \sum_i w_i \right) \tag{7}$$

and each of the sums in the bracket is 1 so that the right hand side is zero. The object of normalisation is a result of a subtlety that has been ignored so far in order to clarify the essentials of the situation. It was assumed the dot product s, gives an indication of the angular separation of the weight and input vectors. This is true up to a point but recall that the dot

product also involves the product of vector lengths. If either the input or weight vectors are large, then s may also be large, not as a result of angular proximity (the vectors being aligned) but simply by virtue of their magnitude. We want a measure of vector alignment which does not require a separate computation of vector lengths, and the normalisation process is one way of achieving this.

Secondly, the learning rule may be expanded to the sum of two terms

$$\Delta \mathbf{w} = \alpha \mathbf{x} y - \alpha \mathbf{w} y \tag{8}$$

The first of these looks like a Hebb term while the second is a weight decay. Thus we may see competitive self-organisation as Hebb learning but with a decay term that guarantees normalisation. This latter property may be thought of in biological terms as a conservation of metabolic resources; thus, the sum of synaptic strengths may not exceed a certain value which is governed by physical characteristics of the cell to support synaptic and post-synaptic activity. There are several architectures that have used the basic principles outlined above. Rumelhart & Zipser (1986) (henceforth R & Z) give a good discussion of competitive learning and several examples. There is only space to discuss one of these here.

2.1 Letter and 'word' recognition

R & Z train using pairs of characters, each one being based on a 7 by 5 pixel grid. In the first set of experiments they used the four letter pairs *AA AB BA BB*. With just two units in a the competitive net, each unit learned to detect either *A* or *B* in a particular serial position. Thus, in some experiments, unit 1 would respond if there was an *A* in the first position while unit 2 would respond if there was a *B* in the first position. Alternatively the two units could respond to the letter in the second position. Note that these are, indeed, the two possible 'natural' pairwise groupings of these letter strings. R & Z call this net a 'letter detector'. With four units each node can learn to respond to each of the four pairs - it is a 'word detector'.

In another set of experiments, R & Z used the letter pairs *AA, AB, AC, AD, BA, BB, BC, BD*. When a net with only two units was used, one unit learned to recognise the pairs which started with *A*, while the other learned to respond to those that began with *B*. When 4 units were used each unit learned to recognise the pairs that ended in one of the four different letters *A, B, C, D*. This represents two different ways of clustering the training set. If the patterns are to be put into two groups then, clearly, it is the first letter which characterises the group. On the other hand, if there are to be four clusters, the four value feature determined by the second letter is the relevant distinction.

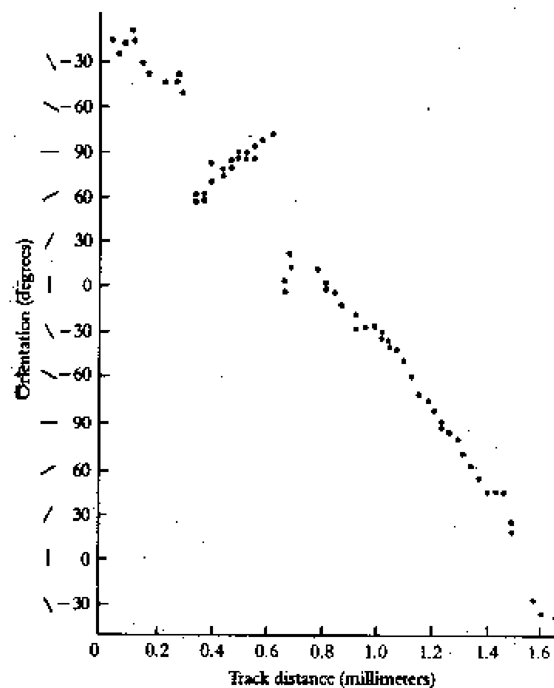
3 Kohonen's self-organising feature maps

3.1 Topographic maps in the visual cortex

It often occurs that sensory inputs may be mapped in such a way that it makes sense to talk of one stimulus being 'close to' another according to some metric property of the stimulus. The simplest example of this occurs when the metric is just the spatial separation of localised sources. A slightly more abstract example is provided by the cells in visual area 1 of the

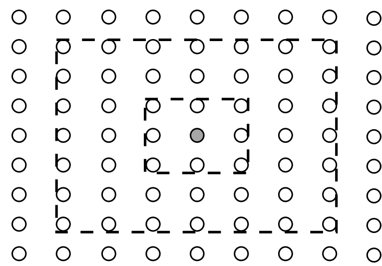
mammalian brain, which are 'tuned' to orientation of the stimulus. That is, if a grid or grating of alternating light and dark lines is presented to the animal, the cell will respond most strongly when the lines are oriented in a particular direction and the response will fall off as the grating is rotated either way from this preferred direction. This was established in the classic work of Hubel & Weisel (1962) using microelectrode studies with cats. Two grating stimuli are now 'close together' if their orientations are similar. This defines a *metric* or measure for the stimuli.

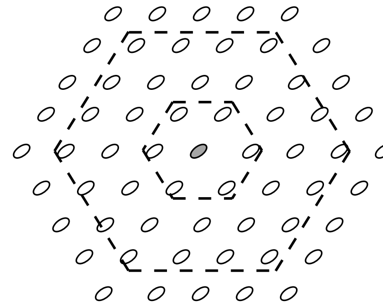
If we were to train a competitive network on a set of gratings then each cell (unit) would learn to recognise a particular orientation. However there is an important property of the way cells are organised in biological nets which will not be captured in our scheme as described so far. That is, cells which are tuned to similar orientations tend to be physically located in proximity with one another. In visual cortex, cells with the same orientation tuning are placed vertically below each other in columns perpendicular to the surface of the cortex. If recordings are made from an electrode which is now inserted parallel to the cortex surface and gradually moved through the tissue, the optimal response from cells will be obtained at a series of orientations that vary, in general, smoothly across the cortex. There are, however occasional discontinuities as shown in the slideorienttrack



slide of data on orientation tuning

The orientation tuning over the surface forms a kind of map with similar tunings being found close to each other. These maps are called *topographic feature maps.* It is possible to train a network using methods based on activity competition in a such a way as to create such maps automatically. This was shown by C. von der Malsburg in 1973 specifically for orientation tuning, but Kohonen (1982) popularised and generalised the method and it is in connection with his name that these nets are usually known. The best exposition is given in his book (Kohonen, 1984). These nets consist of a layer of nodes each of which is connected to all the inputs and which is connected to some neighbourhood of surrounding nodes.





Kohonen neighbourhoods

3.2 The algorithm

At each vector presentation the following sequence of steps occurs

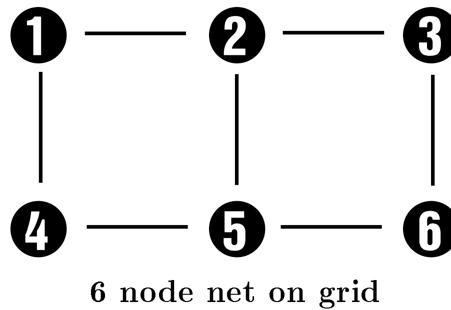
- Find the node k whose weight vector is closest to the current input vector.
- Train node k and all nodes in some neighbourhood of k.
- Decrease the learning rate slightly
- After every M cycles, decrease the size of the neighbourhood

In connection with 1), it is important to realise that Kohonen postulates that this is done with a supervisory computational engine and that his results are not based on the use of competitive dynamics to find the 'winning' node. The justification for this is that it *could* have been done by the net itself with lateral connections. The use of competitive dynamics would slow things down considerably, is very much dependent on the parameters, and does not always work, 'cleanly'. (recall the video demo). In fact the rule Kohonen uses in his examples is to look for the node which simply has the smallest value for the length of the difference vector $\mathbf{x} - \mathbf{w}$. This also appears to obviate the need for input vector normalisation (and hence for the restriction to positive components) which was a prerequisite with the inner product activation measure of proximity. However, this method cannot be the basis of any biologically plausible algorithm.

The key point in this algorithm is 2). It is through this that the topographic mapping arises. It is the use of training over a neighbourhood that ensures that nodes which are close to each other learn similar weight vectors. Decreasing the neighbourhood ensures that progressively finer features or differences are encoded and the gradual lowering of the learn rate ensures stability (otherwise the net may exhibit oscillation of weight vectors between two clusters).

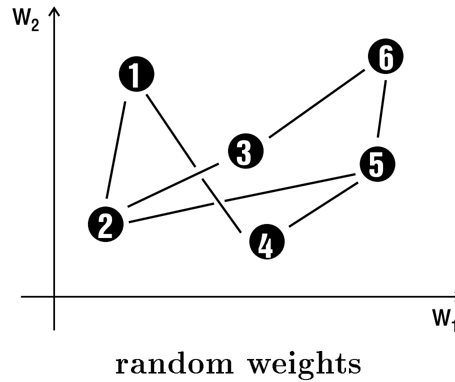
3.3 A graphic example

It is possible to illustrate the self-organisation of a Kohonen net graphically using a net where the input space has just two components. Consider a net with just 6 nodes on a rectangular grid.



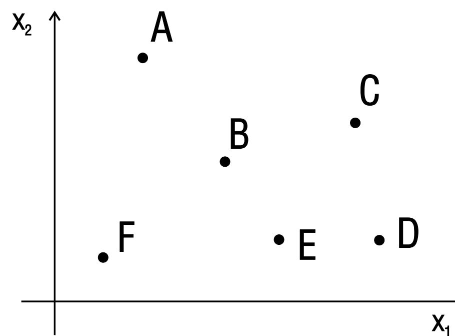
6 node net on grid

Another representation of this net is in *weight space*. Since there are only 2 weights we may draw this on the page. Initially the weights will be random, say



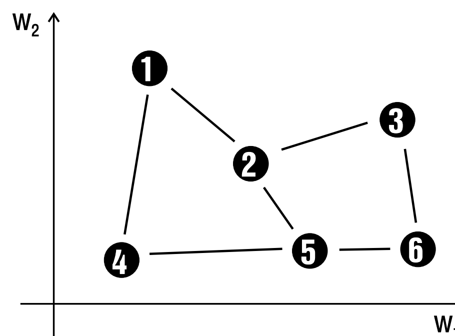
random weights

The lines are drawn to connect nodes which are physically adjacent (first diagram). Suppose now that there are 6 input vectors which may be represented in pattern space as shown below



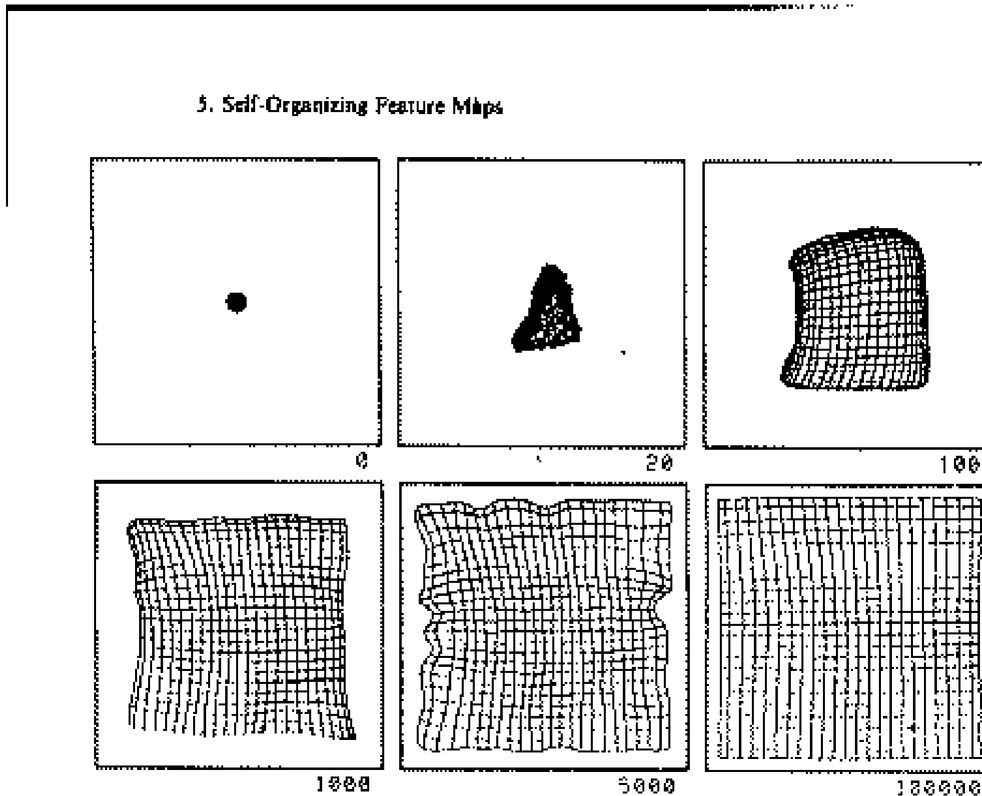
input vectors for 6 node K. net

In a well trained (ordered) net that has developed a topographic map the diagram in weight space should have the same topology as that in physical space and will reflect the properties of the training set.



Trained weight space

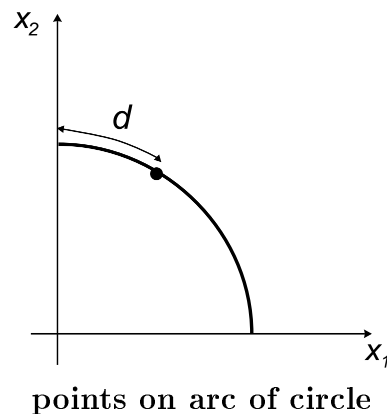
The case of 2-component vectors which are drawn randomly from the unit square and in which the initial weights are close to the centre of the unit square, is dealt with by Kohonen (1984).



slide of kohonens results for 2D square

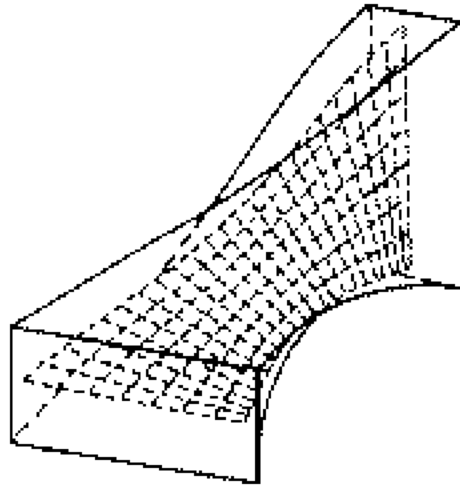
The weight diagram starts as a 'crumpled ball' at the centre and expands like a fishing net being unravelled. Kohonen deals with several other similar examples.

When the input space is more than 2-dimensional, the higher dimensions have to get 'squashed' onto the grid. This will be done in such a way as to preserve the most important variations in the input data. Thus, it is often the case that the underlying dimensionality of the input space is smaller than the number inputs. This is illustrated below for 2-D data which has an underlying dimensionality of 1



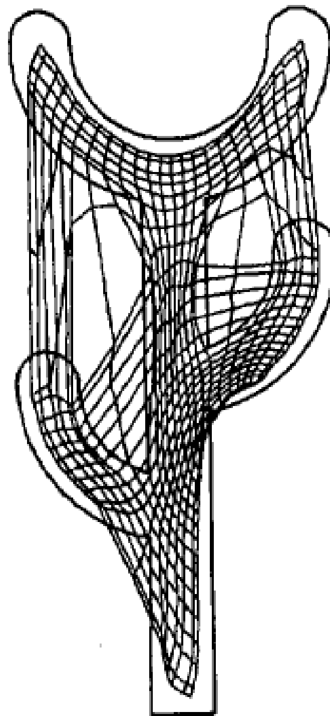
points on arc of circle

The points lie on an arc of a circle and each point may be specified by stating how far round the arc it is. A more convincing example with 3D data with an underlying dimensionalty of 2 is shown in fig 5.9 of Kohonen's book



slide of 3d projection

The topographic map will also refelect the underlying distribution of the input vectors. (Kohonen fig 5.18)



slide of 'cactus' distribution

Returning to the original example of orientation maps in the visual system some of my own recent work has focussed on training nets whose cells form a map of image velocity (Gurney and Wright, 1992)

4 Other competitive nets

Fukushima (1975) has developed a multilayered net called the 'neocognitron' which recognises characters and which is loosely based on early visual processing. The structure is quite complex and, although some of the features seem rather *ad hoc*, it is a very impressive example of modelling a large system which has many similarities with its biological counterpart.

No account of competitive learning would be complete without reference to the work of Stephen Grossberg. His Adaptive Resonance Theory (ART) has been the subject of a enormous number of papers. ART concerns the development of networks in which the number of nodes required for classification is not assigned *ab initio* but is determined by the net's sensitivity to detail within the data set (given by the so-called *vigilance* parameter). The network is embedded in a control loop which is an integral part of the entire system. It would require a complete lecture to do justice to Grossberg's networks however and it will have to suffice here to simply give a reference. This in itself is not easy - Grossberg's work is often quite hard to read and any single reference will undoubtedly be inadequate. One possible route is his 1987 paper in Cognitive Science (Grossberg, 1987).

References

Fukushima, K. (1975). Cognitron: a self-organizing multilayered neural network. *Biological Cybernetics*, 20:121 – 136.

Grossberg, S. (1987). Competitive learning: from interactive activation to adaptive resonance. *Cognitive Science*, 11:23 – 63.

Gurney, K. and Wright, M. (1992). A self-organising neural network model of image velocity encoding. *Biological Cybernetics*, 68:173 – 181.

Hubel, D. and Wiesel, T. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *Journal of Physiology*, 160:106 – 154.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59 – 69.

Kohonen, T. (1984). *Self-organization and associative memory*. Springer Verlag.

Rumelhart, D., McCllelland, J., and The PDP Research Group (1986). Feature disc overy by competitive learning. In *Parallel Distributed Processing*.

von der Malsburg, C. (1973). Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik*, 14:85 – 100.

8: Alternative node types

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

In the first lecture the basic functionality of the most popular artificial neuron - the semilinear unit - was introduced. Thus, the activation a was defined as a linear weighted sum of inputs, $a = \mathbf{w} \cdot \mathbf{x}$, and the output y as the sigmoid of the activation $y = \sigma(a)$. Some variants on this have occurred but the linear-weighted-sum-of-inputs activation has been common to them all. In this lecture, two alternative families of node are introduced which, at first appear dissimilar, but which may be shown to be essentially equivalent. Their possible connection with biological neurons is also outlined.

As a step to introducing alternative types of artificial neuron, note the following features of the semilinear node. The input- output function of the unit is defined in two stages. First, there is an activation which is a function of the inputs x_i and some internal node parameters ζ_j (Greek 'zeta'); that is, $a = a(x_i, \zeta_j)$. For all node types discussed so far the internal parameters are just the weights and the threshold, $\{\zeta_j\} = \{w_k, \theta\}$. The second stage consists of modifying the activation by some non-linearity to produce the final output y; that is $y = g(a)$. So far $g()$ has been either a step function or a sigmoid. This part will be retained - it is the activation which will be modified in what follows.

1 Cubic Nodes

1.1 Using computer memories to generate the activation

Suppose we restrict ourselves to binary inputs which take the values 0 or 1. Consider now the computer memory component shown below.

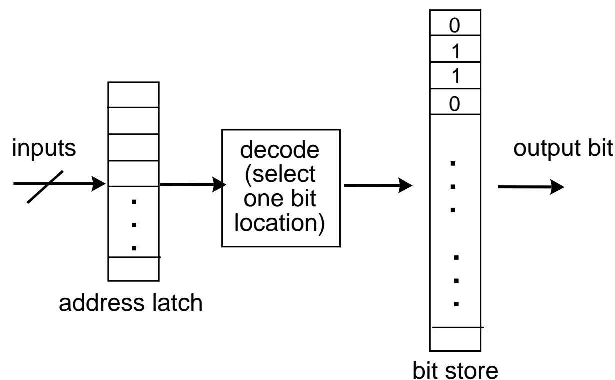
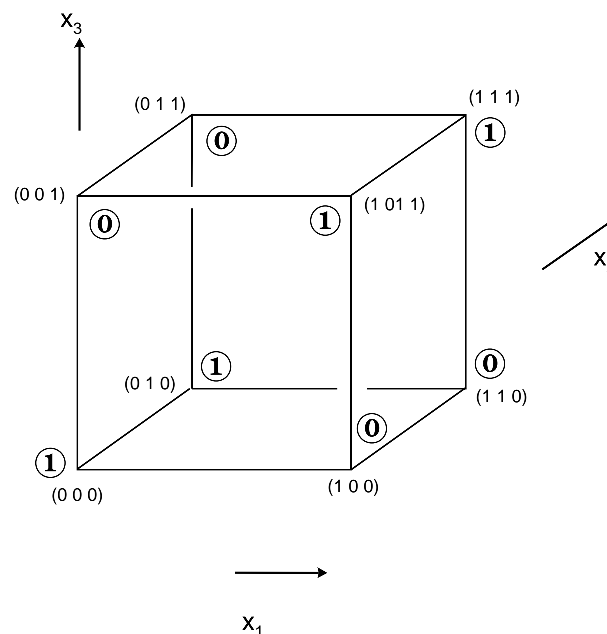


diagram of RAM

Such memories are usually referred to as Random Access Memories or RAMs. * The inputs form an *address* which is decoded to locate one of the memory cells whose contents, either a 0 or a 1, are then read out. If there are n inputs there will be 2^n possible addresses since each location in the address may be either a 0 or a 1. There will therefore be 2^n memory cells whose values, in correspondence with their addresses, may be drawn up in a table similar to the ones used previously to portray the functionality of 2-input TLUs. A 3-input example is shown below.

0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

We may think of the addresses as locating the vertices or *sites* at the corners of the n-dimensional hypercube, just as we did for the TLUs. This is the preferred way of viewing things as it allows a geometric interpretation of generalisation. (next lecture).



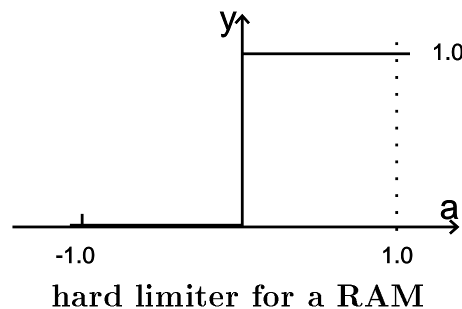
3-cube of table above

However, the key difference now is that we are *not* restricted to those functions which are linearly separable; each RAM location is able to be set to either value independently of the others. This extra functionality may prove useful in solving certain problems and, as discussed later, at least a subset of it may be involved in biological neural processing. The

*This terminology is now largely a misnomer and is rooted in the historical development of computer memories, some of which had to be accessed serially rather than allowing access to arbitrary locations at any time.

further advantage is, of course, the ability to implement our neurons easily in readily available digital hardware since RAMs are a commonplace component in all conventional computers.

In order to accommodate the RAM-lookup table into the general two stage, activation-output scheme described above, it is useful to think of the storage cells as containing the polarised binary values ± 1 (see notes on Hopfield storage prescription).



hard limiter for a RAM

The operation of a RAM as an artificial node now proceeds as follows: A binary input vector is applied to the RAM and is used as the memory address; the (polarised) binary value addressed is read out from the site located and this is then 'converted' back to its unpolarised form using a step function. Within the formalism given above, the internal parameters are now the hypercube site values (memory contents), and the activation is the polarised site value addressed. The site at address μ will be designated S_μ so that $a = a(S_\mu, x_i)$. The function $g(a)$ is then just step function.

RAMs figure strongly in the WISARD machine which was developed at Brunel (Aleksander and Stonham, 1979). The WISARD uses them in a very specific architecture and may be thought of as a hardware implementation of the n-tuple technique developed by Bledsoe & Browning (1959). However, our aim here is to show how the basic node structure described above may be extended and used in conventional network architectures and trained using the well known algorithms. We shall refer to nodes based on the RAM-lookup table model as *cubic nodes* in order to emphasise the geometric view which thinks of them as defined by a population of values at the vertices of the n-cube.

1.2 Writing the activation in closed form

The key that enabled us to develop training algorithms for semilinear nodes was the ability to make explicit the form of the activation

$$a(w_i, x_i) = \sum_{i=1}^{n} w_i x_i \tag{1}$$

That is, it is a 'well-behaved' function of the weights and the inputs which may be manipulated according to normal mathematical procedures. We could then evaluate the gradients, for example, of an error with respect to the weights [†]. At the moment the activation for cubic nodes is just a lookup table - there is no expression corresponding to the right hand side of (1). As a step towards such a closed expression, consider a 2-input cubic node. There

[†]Although we didn't actually do this in the lectures - only plausibility arguments were given - this is the route for a rigorous derivation of BP and the delta rule

are four site values $\{S_{00}, S_{01}, S_{10}, S_{11}\}$, and I now claim that the following is an expression for the activation

$$a = S_{00}(1 - x_1)(1 - x_2) + S_{01}(1 - x_1)x_2 + S_{10}x_1(1 - x_2) + S_{11}x_1x_2 \tag{2}$$

The reason for this is that for any input (address), only one of the product terms in the input variables is non-zero. Thus the expression 'picks out' the relevant site value or activation. Consider for example the input factors in the second term $(1 - x_1)x_2$. This is zero unless $x_1 = 0, x_2 = 1$, that is we have the input (0,1), and then it is just equal to 1. In this case, by similar arguments, all the other input products in the other terms are zero. So we compute $a = S_{01}$ which is what is required.

The expression in (2) may be made more symmetrical in the inputs if we use their polarised form. To highlight this we place a 'bar' over the values so that the inputs are now denoted by $(\bar{x}_1, \bar{x}_2)$. The relation between the two forms is

$$x_i = \frac{1}{2}(\bar{x}_i + 1) \tag{3}$$

Then substituting this in (2)

$$\begin{aligned} a \quad = \quad & S_{00}\frac{(1 - \bar{x}_1)}{2}\frac{(1 - \bar{x}_2)}{2} + S_{01}\frac{(1 - \bar{x}_1)}{2}\frac{(1 + \bar{x}_2)}{2} + \\ & S_{10}\frac{(1 + \bar{x}_1)}{2}\frac{(1 - \bar{x}_2)}{2} + S_{11}\frac{(1 + \bar{x}_1)}{2}\frac{(1 + \bar{x}_2)}{2} \end{aligned} \tag{4}$$

The pattern of signs in the brackets is clearly determined by the pattern of 1s and 0s in the address μ of the site associated with these terms. Thus, if the address μ is written out in terms of its polarised components as $\bar{\mu}_1\bar{\mu}_2$, then the general term in (4) is given by

$$S_\mu \frac{(1 + \bar{\mu}_1\bar{x}_1)}{2}\frac{(1 + \bar{\mu}_2\bar{x}_2)}{2} \tag{5}$$

In the general n-input case we have

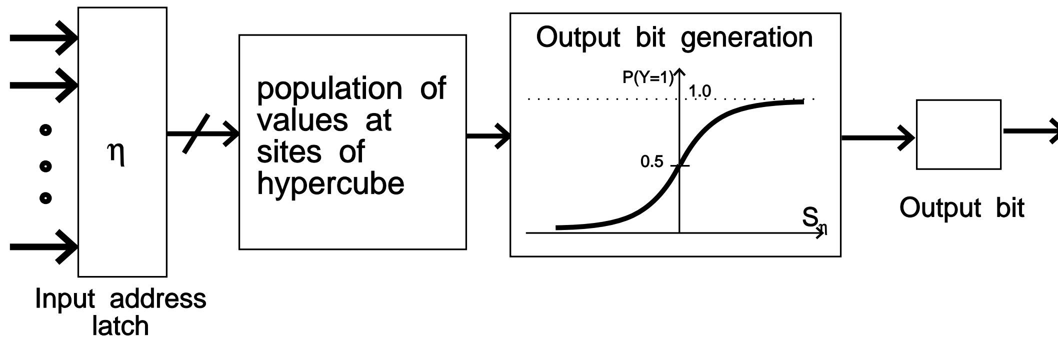
$$a = \frac{1}{2^n}\sum_\mu S_\mu \prod_{i=1}^{n}(1 + \bar{\mu}_i\bar{x}_i) \tag{6}$$

The symbol Π (greek upper case 'pi') means 'product' and all terms under it are multiplied together.

1.3 Training cubic nodes: loss of information and its solution

Cubic nodes get trained by changing their site values. Let '+' denote an increment operation - change the site value to 1, and '-' a decrement operation - change the site value to -1. Suppose a sequence of training steps occurs for a site of the form $+ + - + + + + + + -$, that is 8 increments and 2 decrements. The final value of the site is given by the last training operation, in this case a -1. However, it is clear that training is trying, on average, to increment the site; we are losing information. The solution - as noted as long ago as 1962 by Bledsoe & Blisson (Bledsoe and Blisson, 1962) - is to allow the site to take on more than two values. Thus, we allow sites to take values in the range $(-S_m, -S_m + 1, \ldots, -1, 0, 1, \ldots S_m - 1, S_m)$. Now,

applying the same training sequence to a site with $S_m = 10$, starting with $S_\mu = 0$ we get the following values for the site as it is trained: 1,2,1,2,3,4,5,6,7,6. The output function needs to reflect this flexibility and we now make use of a sigmoid rather than the hard-limiter.



Full cubic node with sigmoid output

Of course, to work with these nodes mathematically it is necessary to assume that the sites are continuous, just as the weights were for semilinear nodes. Only then does it make sense to take gradients, etc. with respect to the site values. In any implementation however, we would allow only discrete site values as described above, so that we may take advantage of RAM technology. Using site values that may only take on a set of discrete values introduces noise into learning since the exactly computed site changes cannot be made.

1.4 Working with analogue inputs

Clearly we can't simply apply non-binary numbers to the inputs of a cubic node; there has to be a binary address in order to locate one of the sites on the cube. The way out is to communicate analogue values by stochastic bit streams (recall stochastic semilinear nodes). The result is that the $\bar{x}_i$ get reinterpreted as 'polarised probabilities' (simply probabilities rescaled to the interval [-1, 1]). Full details may be found in (Gurney, 1992a; Gurney, 1992b).

2 Sigma-Pi nodes

Consider a 3 input semilinear unit. Its activation is

$$a = w_1 x_1 + w_2 x_2 + w_3 x_3 \tag{7}$$

This supposes that each input contributes to the activation independently of the others. That is, the contribution to the activation from input 1 say, is always a constant multiplier (w_1) times x_1. Suppose however, that the contribution from input 1 depends also on input 2 and that, the larger input 2, the larger is input 1's contribution. The simplest way of modelling this is to include a term in the activation like $w_{12}x_1x_2$ where $w_{12} > 0$ (for a diminishing influence of input 2 we would, of course, have $w_{12} < 0$). In general we might have terms containing all possible pairs of inputs and also a term in the three inputs together

$$\begin{aligned} a \;=\; & w_1 x_1 + w_2 x_2 + w_3 x_3 + \\ & w_{12} x_1 x_2 + w13 x_1 x_3 + w_{23} x_2 x_3 + \\ & w_{123} x_1 x_2 x_3 \end{aligned} \tag{8}$$

In general for n-inputs

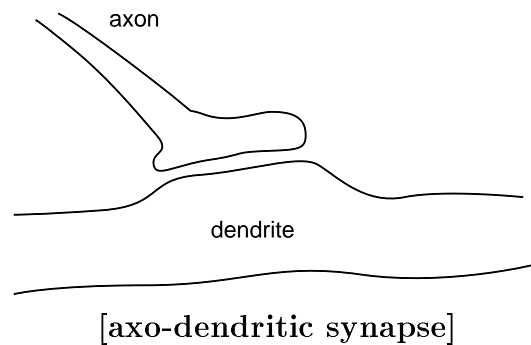
$$a = \sum_{k=1}^{N} w_k \prod_{i \in I_k} x_i \tag{9}$$

where I_k is the kth in a series of index sets, each of which contains one of the possible 2^n selections of the first n integers. There are therefore 2^n 'weights' in this type of node. The presence of the 'sigma' and 'pi' symbols together here gives rise to the term *sigma-pi* units. (PDP vol 1 page 72-74). Now, although the terms contain products of inputs, there are no powers of each input greater than one; this gives rise to the name *multilinear* for the terms in this kind of expression. Nodes with multilnear terms are also sometimes called *higher- order* nodes, since their activation depends on terms whose multiplicative order is greater than one.

Comparison of (9) and (6) shows that they are superficially similar. In fact, after a little manipulation, the latter expression may be cast in exactly the same form as the first. Thus, cubic nodes may be thought of as a kind of sigma-pi node (Gurney, 1992b).

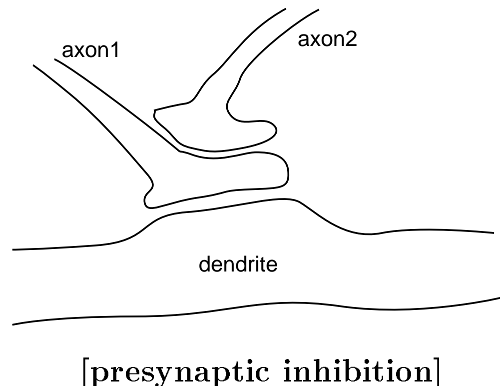
3 Biological neurons and sigma-pi units

The stereotypical synapse shown below is the inspiration for most neural net modelling and was introduced in lecture 1. It consists of an electro-chemical connection between an axon and a dendrite - hence it is an *axo-dendritic* synapse



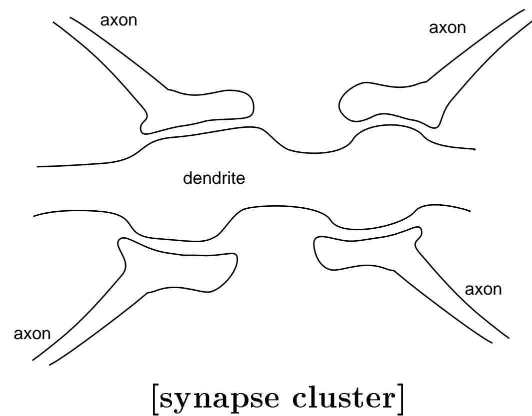
[axo-dendritic synapse]

However there is a large variety of synaptic types and connection grouping (See for example, Dayhoff ch. 8 for a review). Of special importance here are the pre-synaptic inhibitory synapses of the serial variety and clusters of densely packed axo- dendritic synapses



[presynaptic inhibition]

Here the efficacy of the axo-dendritic synapse between axon 1 and the dendrite is modulated (inhibited) by the activity in axon 2 via the axo-axonic synapse between the two axons. This might therefore be modelled by a quadratic term like $w_{12}x_1x_2$



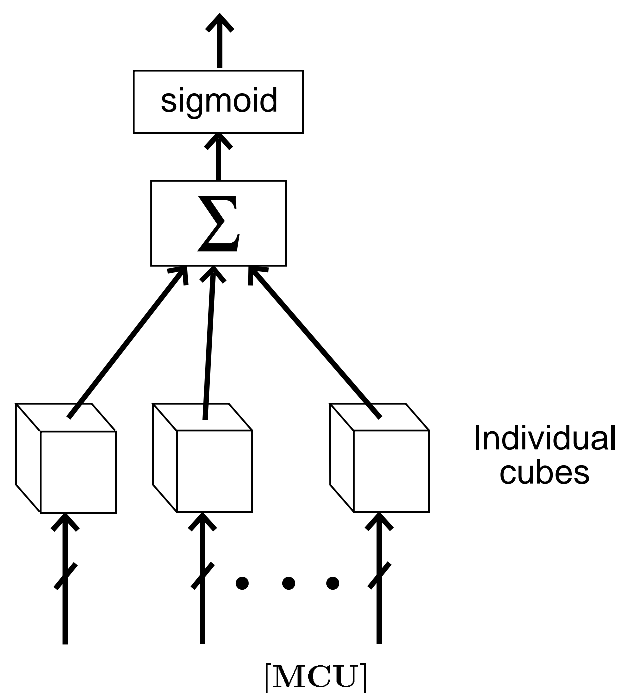
[synapse cluster]

Here the effect of the individual synapses will surely not be independent and we should look to model this with a multilinear term in all the inputs.

4 Pruning sigma-pi units: the multi-cube node

Consider a cubic node with n-inputs. There are 2^n sites. For $n = 8$ this gives 256 sites. For $n = 32$ this gives $2^{32} \approx 10^9$ sites. Clearly there is an explosion of the number of sites as n grows which is not only impractical from an implementation point of view, but also suspect from a theoretical viewpoint: do we really need to make use of all the functional possibilities available in such nodes? (remember problem sheet 2, and how the number of functions increases as the number of inputs).

One way of overcoming this is developed in the *Multi-cube unit* or MCU (Gurney, 1992a; Gurney, 1992b) where several small cubes sum their outputs to form the activation.



[MCU]

This also has a biological analogue in relation to the synaptic clusters described above. The small cubes are supposed to correspond to these clusters which then sum their contributions linearly. Mathematically, in terms of the sigma-pi form, we are limiting the order of terms that are used. So for example, if the cubes all have just 4 inputs, there can only be terms containing, at the most, products of 4 inputs.

5 The terminological Tower of Babel...

I have used the term 'cubic node' to denote one in which the activation is found by looking up the value at the corner of a hypercube. I do this to give an implementation-free viewpoint although, as we have seen, the nodes are equivalent to a RAM lookup with several bits stored at each memory location. Because of Aleksander's emphasis on the hardware, he and his group accent the RAM-based approach and talk of RAM-nets. Further, when talking about the multivalued site nodes with sigmoidal output, they talk of Probabilistic Logic Nodes or PLNs. Aleksander has also tried recently to emphasise the distinction to be made with normal weighted units by referring to them as 'weightless' nodes. However, the sigma-pi equivalence that I have noted would lead me to counter this by saying that they are far from weightless! John Taylor has his own set of acronyms which is centered on that for 'probabilistic RAM' or pRAM'.

There was almost a consensus about 3 years ago when everyone agreed to call them generically 'digital nodes' since they could be implemented using digital hardware. This was pretty sensible and therefore seems to have been abandoned!

A more complete exposition of the cube based (implementation-free) approach may be found in (Gurney, 1989) which is available in a (physically) more compact form as a Technical Memo.

References

Aleksander, I. and Stonham, T. (1979). Guide to pattern recognistion using random-access memories. *Computers and digital techniques*, 2:29 – 40.

Bledsoe, W. and Blisson, C. (1962). Improved memory matrices for the n-tupel pattern recognition method. *IRE Transactions on Electronic Computers*, EC11:414 – 415.

Bledsoe, W. and Browning, I. (1959). Pattern recognition and reading by machines. In *Proceedings of the Eastern Joint Computer Conference*, pages 225 – 232.

Gurney, K. (1989). *Learning in networks of structured hypercubes.* PhD thesis, Dept. Electrical Engineering, Brunel University, Uxbridge, Middx, UK. Available as Technical Memorandum CN/R/144.

Gurney, K. (1992a). Training nets of hardware realisable sigma-pi units. *Neural Networks*, 5:289 – 303.

Gurney, K. (1992b). Weighted nodes and ram-nets: A unified approach. *Journal of Intelligent Systems*, 2:155 – 186.

9: Cubic nodes (contd.) and Reward Penalty training

Kevin Gurney

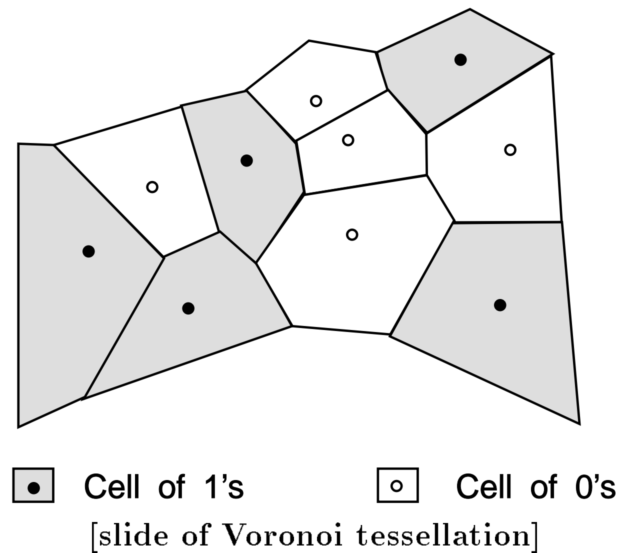
Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

This lecture deals with training nets of cubic nodes and introduces another major (quite general) algorithm - Reward Penalty. Insight into how we might train nets of cubic nodes is provided by considering the problems associated with generalisation in these nets. We then go on to consider feedback or recurrent nets from the point of view of their implementing iterated feedforward nets (recall this discussion in the case of Hopfield nets). Although the discussion here centres on cubic nodes, it provides insight into recurrent nets quite generally. Reward Penalty is introduced and is shown to apply to cubic as well as semilinear nodes.

1 Generalisation in Cubic Nodes - centres and clustering

First, recall the action of TLUs for comparison. The operation of an n-input TLU on Boolean vectors is determined by a hyperplane passing through the n-cube, so that all vectors on one side of this plane will produce a '1', while the others generate a '0'. Suppose a TLU has been trained to classify two input vectors. Every other possible input pattern will now be classified according to the node's hyperplane and there is automatic generalisation across the whole input space. (Recall the training of a 2-input TLU with only 2 vectors).

Consider now, a cubic node which, in the untrained state, has all sites set to zero. The output to any vector will be totally random with there being equal probability of a 1 or a 0. If this node is now trained on two (Boolean) vectors, only the two sites addressed by these will have their values altered; any other vector will produce a random output and there has been no generalisation. We shall call sites addressed by the training set *centre sites* or *centres*. In order to promote Hamming distance generalisation, sites close to the centres need to be trained to the same or similar value as the centres themselves. That is, there should be a *clustering* of site values around the centres. This is true in both feedforward and recurrent cube- based nets and there are various ways of achieving this which are discussed later. The situation is shown schematically in the diagram attached.



[slide of Voronoi tessellation]

Here, the extreme case is shown where the entire cube has been partitioned and no sites remain untrained. This algorithm for doing this has led to a *Voronoi* tessellation of cube, where the value a site takes on is determined by the value of its nearest centre; sites equidistant from opposite centres remain at 0. Training can now be seen as a two stage process; first centre sites have to be established according to the training set and then clusters developed around them. Before looking at ways of doing this, however, it is instructive to look more closely at the way clustering may help recurrent nets in auto-associative pattern recall. *

It is useful at this stage to summarise certain aspects of cubics nodes by making a comparison with the more usual semilinear weighted variety

- Cubic nodes allow greater functionality than semilinear nodes and therefore may be expected to implement some problems with fewer nodes.
- Cubic nodes have a ready implementation in RAM
- Cubic nodes do not exhibit intrinsic generalisation; they must be coerced to do so. Compare with linear dichotomy of TLUs
- The resources required for cubic nodes increases exponentially (number of site values varies like 2^n). May require use of MCUs.

2 Recurrent nets of cubic nodes

Although the discussion here centres on cubic nodes, it provides insight into recurrent nets quite generally.

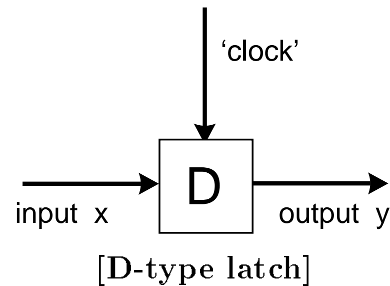
3 The Action of Well-Trained Recurrent Nets

We will limit the discussion here to the case where the unit output function is a hard limiter or very steep sigmoid so that any non-zero site value gives rise to a deterministic output. This

*Just as in the Hopfield nets in the 'A', 'B' simulation demo.

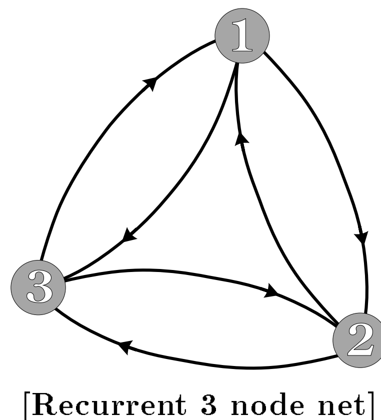
clarifies the argument which may later be qualified to take into account any probabilistic behaviour. The dynamics are also supposed, in the first instance, to be synchronous (all nodes update at the same time).

It is useful to think of a recurrent net performing pattern recall as iterating an input-output relationship. This is made explicit by 'unfolding' the net into a feedforward one, obtained by breaking the feedback loops and sending signals through a sequential iteration of hardware rather than through the same physical units. This was alluded to in the lectures on Hopfield nets but is restated here in the specific context of synchronous dynamics. A key component here is the use of delay or buffering elements (or temporary storage/latches) to avoid signal 'racing'. These are shown below



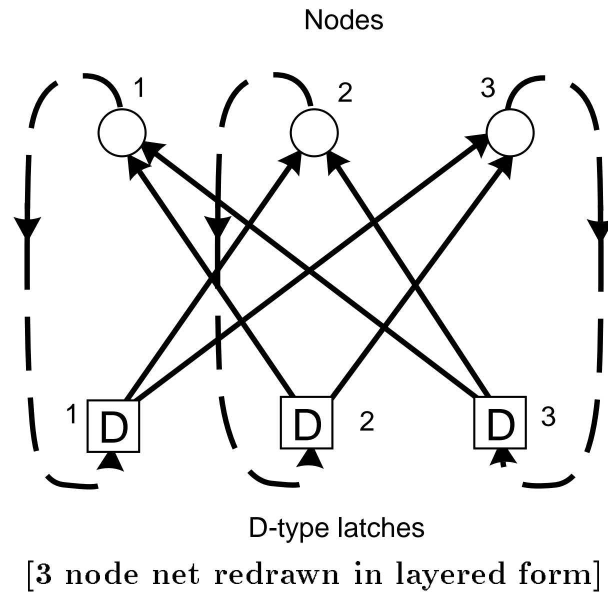
[D-type latch]

Their operation is very simple. Upon the application of a 'store' or 'clock' signal, the output of the device becomes equal to its input. Any alteration of the input has no affect on the output until another clocking signal is received. Now consider the (usual) 3 node network shown below.



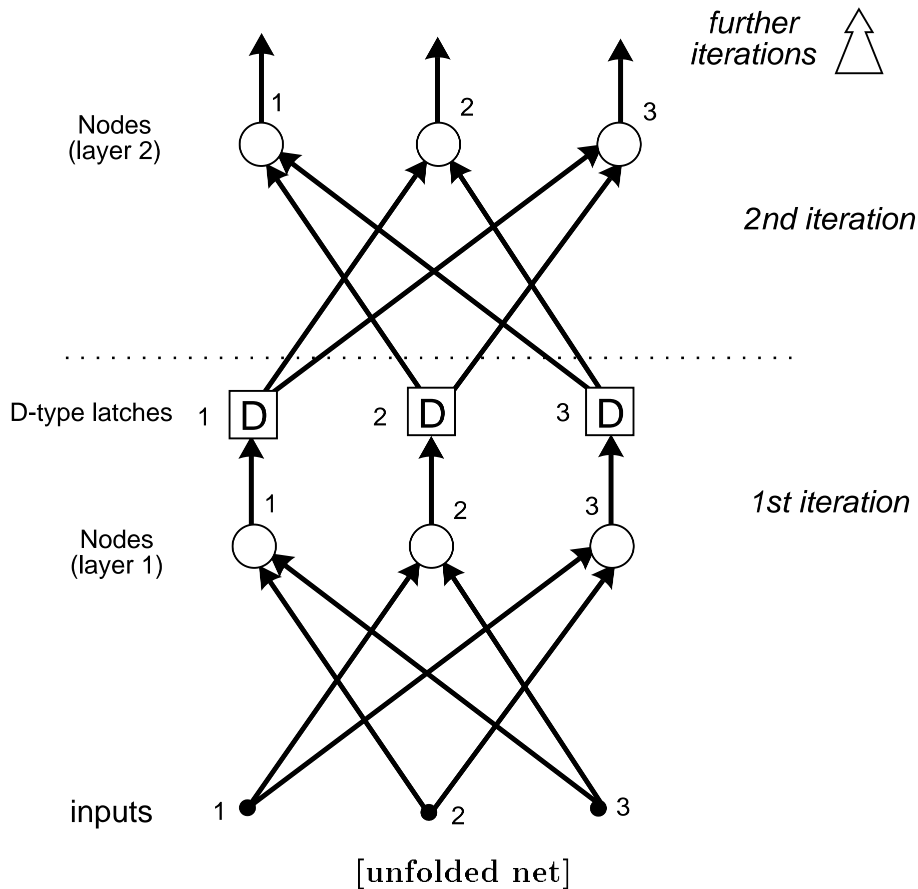
[Recurrent 3 node net]

In order to run this synchronously, we must store the previous state while evaluating the next state of the net. This may be done using a set of D-type temporary storage elements. These are not shown explicitly in the above representation but are shown in the network redrawn as a layered structure below.



[3 node net redrawn in layered form]

The state of the net is given by the contents of the Delay elements. To find the next state, the nodes process their inputs from the D-types, output the new values and then the D-type are 'clocked' to store the new state. This process is iterated indefinitely or until a single-state cycle has been reached. This iteration may be done using the same physical hardware, by feeding back the output as shown above, or by feeding the output forward into a replica of the same hardware as shown below.



[unfolded net]

The processing which takes place in the kth time step in the recurrent net now takes place at the kth layer of the feedforward net.

We now consider the processing of an initial state **u** ('input' in the unfolded net) which is close in Hamming distance to a trained state cycle **v**. Suppose the net has well clustered nodes; that is each centre is surrounded by a cluster of similar valued sites to the centre value itself. In the ideal case, starting from a pattern **u** close to a member **v** of the net's training set, the latter is recalled after one time step (single layer in the unfolded net). This happens because, although some of the sites addressed in some nodes are not centres, they are still sufficiently close to the desired centres to fall within the clusters associated with them and therefore give the same value as the centres themselves. In general, however, it will take several transitions (layers) to do this since some addressed sites will fall outside the relevant clusters. Thus, after one transition (layer), **u** will have been transformed into $\mathbf{u}'$, where the latter has more components in common with **v** than **u**. This has occurred because enough sites addressed by **u** are in the clusters associated with the centres of **v** and hence have a similar value. In subsequent transitions (layers) the same occurs but with fewer sites being visited which are outside the relevant clusters (or those of the correct value). The number of these erroneous sites should decrease to zero as the net moves through state-space.

A similar argument may be applied to the operation of recurrent nets of TLUs (e.g. Hopfield nets) but here the clusters are restricted to two regions of the n-cube which are linearly separable.

3.1 Training recurrent nets

3.1.1 Training centres

It is a simple matter to enforce a state cycle in a feedback network. We force or *clamp* the Delay latches to the desired state and train each node so that its output is just equal to the value on its corresponding latch. The sites addressed during training are, of course, centre sites. Note that there may be conflict between the training required for different vectors. This will occur when there is a component subset of a vector, forming the address to a node, which is the same as the corresponding subset of another vector, and the node is supposed to give opposite outputs in each case. For example, we couldn't train the 3-node net to the two states (001) and (000) since node 3 is required to give opposite outputs in each case to the address (00). To a lesser degree, the problem will become manifest if there are centres of opposite value which are close together. This leads to small clusters around these centres and, as the problem grows, the cube becomes 'fragmented'. These difficulties may be overcome in the following way.

Suppose that we add units to the net. It may now be possible to augment the original training vectors to include components on the new units so that centre value conflicts are removed and fragmentation reduced. Thus, in the 3-node net, adding a hidden node (labelled 4, say) which takes on different values for each of the conflict-producing vectors, will allow the two (augmented) vectors (001*0*) and (000*1*) to be learnt (the values of node 4 are in italics). These units are properly regarded as *hidden* since we are not interested in their vector components *per se* and may not impose these from outside; rather, the net should 'discover' them for itself. In terms of state-space, hidden units are being used to prevent overlap of centres of attraction and helping make each basin of attraction as large as possible.

This process of centre optimisation may be likened to a physical process where centres

are allowed to move over the vertices of the cube and are subject to inter-centre forces. By making unlike centres repel we aim to avoid cube fragmentation and conflict. As in a physical system, the forces may be defined via a potential energy function. The problem then reduces to a function minimisation and may be tackled with simulated annealing. Further details may be found in (Gurney, 1989) together with simulation examples.

3.1.2 Developing site clusters

The most straightforward way to do this is to suppose that the net is endowed with enough processing ability to perform the Voronoi tessellation algorithmically. This will result in a net with site values $\pm S_m$ within cells and 0 at any sites on cell boundaries. This is an *off-line* technique in that it is not naturally done by the intrinsic neural hardware during training. Further details may be found in (Gurney, 1989; Gurney, 1992b). Aleksander (1991) has independently suggested a similar technique and embodied it in the so-called gRAM node.

Another possibility is to present noisy copies of the training set, thereby visiting sites close to true centres. This has been the basis of methods developed by Milligan (1988) who describes the process as one of modifying the net's state-space structure. It may be done naturally using an enhancement of the node's architecture described in (Gurney, 1992a).

4 Reward Penalty training

Consider a single node which has stochastic output. It may be semilinear or cubic, the only requirement is that it gives a boolean output which depends on the activation stochastically according to some nonlinear law like the sigmoid. Suppose that, just as for the delta rule, there is a set of input-output pairs so that each vector is associated with a 0 or 1. On applying a vector and noting the output, we compute a signal which is '1' ('Reward') if the node classified the input correctly, and '0' ('Penalty') if it classified incorrectly. If the node is rewarded, it adjusts its internal parameters (weights or site values) so that the current output is *more* likely with the current input. If, on the other hand, the node is penalised, it adjusts its parameters so that the current output is less likely. Specifically, for a stochastic semilinear node, the change in the weight Δw_i on input i

$$\Delta w_i = \begin{cases} \alpha[y - \sigma(a)]x_i & \text{if} \quad r = 1 \\ \lambda\alpha[1 - y - \sigma(a)]x_i & \text{if} \quad r = 0 \end{cases} \tag{1}$$

Here α is a learning rate, as usual, and λ is a constant which governs the relative importance of penalising. Empirically, λ values of around 0.2 appear most successful.

It can be shown (Williams, 1987) that this leads to a stochastic or noisy gradient descent. We are using less information than with the delta rule where we explicitly calculated the error gradient. This means that training will take longer. However, the training rule works for both hidden and output nodes directly without any modification. Recall that there was a substantial amount of calculation involved in evaluating the δ's in backpropagation. There is therefore a tradeoff between the complexity of each training step and the number of steps needed. Reward-Penalty (RP) also has the benefit that there is some noise which may be useful in evading entrapment in local minima. When there is more than one output node

it is convenient to define an error e which is normalised to the range [0,1] and then define the probability of rewarding as $1 - e$

Barto and Jordan (1987) have successfully used RP in training nets of semilinear nodes. In fact, they train the output layer with the delta rule and the hidden layer with RP. This is sensible since the amount of computational overhead in the delta rule is small and it greatly increases the rate of learning overall. I (Gurney, 1989; Gurney, 1992a) showed that the above learning rule led to convergence for cubic nodes by adapting the proof of Williams (1987). Aleksander and Myers (1988) have developed an algorithm with an RP 'flavour' which they use to train nets of PLNs. Work is actively being pursued in developing hardware to train nets of MCUs using this algorithm (Hui et al., 1993; Bolouri et al., 1994) and the first generation of such chips has now been fabricated and tested (see poster in prefab 3).

References

Aleksander, I. (1991). (not sure of chapter title). In Eckmiller and Hartmann, editors, *Parallel Processing in Neural Systems and Computers,*, pages 2 – 5. North Holland.

Barto, A. and Jordan, M. (1987). Gradient following without backpropagation in layered networks. In *1st Int. Conference Neural Nets, San Diego*, volume 2.
(I have this).

Bolouri, H., Morgan, P., and Gurney, K. (1994). Design, manufacture and evaluation of a scalable high-performance neural system. *Electronics Letters*, 30:426.

Gurney, K. (1989). *Learning in networks of structured hypercubes.* PhD thesis, Dept. Electrical Engineering, Brunel University, Uxbridge, Middx, UK. Available as Technical Memorandum CN/R/144.

Gurney, K. (1992a). Training nets of hardware realisable sigma-pi units. *Neural Networks*, 5:289 – 303.

Gurney, K. (1992b). Training recurrent nets of hardware realisable sigma-pi units. *International Journal of Neural Systems*, 3:31 – 42.

Hui, T., Morgan, P., Gurney, K., and Bolouri, H. (1993). A cascadable 2048-neuron VLSI artificial neural network with on-board learning. In Aleksander and Taylor, editors, *Artificial Neural Networks 2*, pages 647 – 651. Elsevier.

Milligan, D. (1988). Annealing in ram-based learning networks. Technical Report CN/R/142, Dept. Electrical Engineering, Brunel University.

Myers, C. and Aleksander, I. (1988). Learning algorithms for probabilistic neural nets. In *1st INNS Annual Meeting.*

Williams, R. (1987). Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3, Northeastern University Boston.

10: Drawing things together - some perspectives

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx.

It is the intention of this final part to look back on the technical material and try to perceive some overall structure. This is done principally by developing a neural net taxonomy or classification scheme but also by providing historical and disciplinary perspectives.

1 Nets vs. Symbols (revisited)

This was the point of departure at the beginning of the course but we review the two paradigms again, this time in more depth and with some historical background.

From the early days of computing in the late 1940s and early '50s, there have existed two approaches to the problem of developing machines which exhibit 'intelligent' behaviour. One of these tries to capture knowledge in some domain as a set of atomic semantic objects or *symbols*, and to manipulate these according to a set of formal algorithmic rules. This symbolic- algorithmic paradigm has, over the last twenty years, represented the mainstream of research in Artificial Intelligence, and indeed the very term 'AI' is usually taken to refer to this school of thought.

Concurrent with this however, has been another line of research which uses machines whose architecture is loosely based on that of the animal brain, and which learn from a training environment, rather than being preprogrammed in some high level computer language. Work with these so-called *neural networks* was very active in the 1960s, suffered a loss of popularity, during the '70s and early '80s, but is now enjoying a revival of interest.

1.1 The symbolic paradigm

Although the first electronic digital computers were designed to perform numerical calculations, it was apparent to the early workers that the machines they had built were also capable of manipulating symbols, since the machines themselves knew nothing of the semantics of the bit-strings stored in their memories. Thus Alan Turing speaking in 1947 about the design for the proposed Automatic Computing Engine (ACE), saw the potential to deal with complex game playing situations 'Given a position in chess the machine could be made to list all the "winning combinations" to a depth of about three moves....' quoted from [18].

The machines on which the modern AI fraternity now run their algorithms have not changed in any fundamental conceptual way from the Pilot ACE which was eventually built; all of these being examples of the classic Von Neumann architecture. Granted, there has been a speed increase of several orders of magnitude, and hardware parallelism is sometimes available, but contemporary 'AI engines' are still vehicles for the instantiation of the theoretic stance which claims that cognition can be described completely as a process of formal, algorithmic symbol manipulation.

Mainstream AI has proved successful in many areas and, indeed, with the advent of expert systems has become big business. For a brief history of its more noteworthy achievements see [29].

However AI has not fulfilled much of the early promise that was conjectured by the pioneers in the field. This is brought home by Dreyfus in his book 'What Computers Can't Do' [8] where he criticizes the early extravagant claims and outlines the assumptions made by AI's practitioners. Principal among these are the belief that all knowledge or information can be formalised, and that the mind can be viewed as a device that operates on information according to formal rules. It is precisely in those domains of experience where it has proved extremely difficult to formalise the environment, that the 'brittle' rule-based procedures of AI have failed.

The differentiation of knowledge into that which can be treated formally and that which cannot, is made explicit by Smolensky (1988) [34] where he makes the distinction between *cultural*, or *public knowledge* and *private*, or *intuitive knowledge*. The stereotypical examples of the former are found in science and mathematics, whereas the latter describes, for instance, the skills of a native speaker or the intuitive knowledge of an expert in some field. In the *connectionist* view, intuitive knowledge cannot be captured in a set of linguistically formalized rules and a completely different strategy must be adopted.

1.2 The connectionist paradigm

The central idea is that in order to recreate some of processing capabilities of the brain it is necessary to recreate some of its architectural features. Thus a connectionist machine, or *neural net*, will consist of a highly interconnected network of comparatively simple processors (the *nodes*, *units* or *artificial neurons*) each of which has a large fan-in and fan-out. In biological neurons the distinctive processing ability of each neuron is supposed to reside in the electro-chemical characteristics of the inter-neuron connections, or synapses. In many connectionist systems this is modeled by assigning a connection strength or *weight* to each input. However, there are other ways of associating a set of parameters to a node which capture its functionality as in, for example, the cubic nodes. In all cases the net ensemble of these is the embodiment of the knowledge the system possesses.

Moving to dynamics, biological neurons communicate by the transmission of electrical impulses, all of which are essentially identical, so that information is contained in the spatio-temporal relationships between them. Neurons are continually summing or *integrating* the effects of all its incoming pulses and, depending on whether the result is excitatory or inhibitory, an output pulse may or may not be generated. In artificial nets, each node continually updates its state by generating an internal *activation value* which is a function of its inputs and internal parameters. This is then used to generate an output via some activation-output function which is typically a squashing function like the sigmoid.

At the system level, it is possible to draw up a list of features displayed by many artificial neural nets but, since the connectionist banner has been attached to such a wide diversity of systems, any such list will inevitably not apply to all of them; however the following is a typical set of characteristics:

- The node parameters are trained to their final values by continually presenting members from a set of patterns or *training vectors* to the net, allowing the net to respond to each presentation, and altering the weights accordingly; that is, they are adaptive rather than preprogrammed systems.

- Their action under presentation of input is often best thought of as the time evolution of a dynamical physical system and there may even be an explicit description of this in terms of a set of differential equations. This was the nature, for example, of the continuous valued Hopfield net and the competitive nets.

- They are robust under the presence of noise on the inter-unit signal paths and exhibit graceful degradation under hardware failure. (Recall the examples on the 1st problem sheet)

- A characteristic feature of their operation is that they work by extracting statistical regularities or *features* from the training set. This allows the net to respond to novel inputs in a useful way by classifying them with one of the previously seen patterns or by assigning them to new classes. (Recall the simulation exercise with the 11-input TLU)

- Typical modes of operation are as associative memories, retrieving complete patterns from partial data (Recall the Hopfield net demo), and as pattern classifiers (The A/B classification with delta rule simulation)

- There is no simple correspondence between nodes and high level semantic objects. Rather, the representation of a 'concept' or 'idea' within the net is via the complete vector of unit activities, being distributed over the net as a whole, so that any given node may partake in many semantic representations. Think about the 'codes' in the 424 encoders etc. which form distributed (internal) representations of the vectors which are then explicitly represented in a one-to-one way at the outputs.

Concerning this last point there is a divergence of opinion in the connectionist camp. Many workers have exhibited nets that contain at least some nodes which denote high level semantic constructs [32, 37]. In [34] Smolensky argues for the 'Proper treatment of Connectionism', in which nets can only operate at a *sub-symbolic* level and where there are no local high- level semantic representations. He notes that, otherwise, connectionism runs the risk of degenerating into a parallel implementation of the symbolic paradigm. Indeed, high level semantic nets have been studied outside the connectionist milieu as 'pulse networks' on weighted digraphs [7] where no 'neural' analogy is implied.

2 A brief history of neural nets

2.1 The early years

The ideas concerning machines that incorporate neural features have always been contemporaneous with work on the general purpose computers in common use today. In fact the analogy between computing and the operation of the brain was to the fore in much of the early work in this area. Thus von Neumann, in the first draft of a report on the EDVAC [35], makes several correspondences between the proposed circuit elements and animal neurons.

In 1942 Norbert Weiner [15] and his colleagues were formulating the ideas that were later christened 'Cybernetics', and which he defined as 'control and communication in the animal and the machine'. Central to this programme, as the description suggests, is the idea that biological mechanisms can be treated from an engineering and mathematical perspective. Of central importance here is the idea of *feedback*

[demo xpt on feedback - 'fingers together']

With the rise of AI and cognitive science, the term 'Cybernetics' has become unfashionable in recent years [1] it might be argued that, because of its interdisciplinary nature, the new- wave of connectionism should properly be called a branch of Cybernetics: certainly many of the early neural-net scientists would have described their activities in this way.

In the same year that Weiner was formulating Cybernetics, McCulloch and Pitts [26] published the first formal treatment of artificial neural nets. The main result in this historic (but opaque) paper, as summarized in [36] is that any well defined input-output relation can be implemented in a formal neural network.

One of the key attributes of nets is that they can learn from their experience in a training environment. In 1949, Donald Hebb [13] indicated a mechanism whereby this may come about in real animal brains. Essentially, synaptic strengths change so as to reinforce any simultaneous correspondence of activity levels between the pre-synaptic and post-synaptic neurons. Translated into the language of artificial neural nets, the weight on an input should be augmented to reflect the correlation between the input and the unit's output. Learning schemes based on this 'Hebb rule' have always played a prominent role.

The next milestone is probably the invention of the *Perceptron* by Rosenblatt in 1957; much of the work with these is described in his book 'Principles of Neurodynamics' [30]. One of the most significant results presented there was the proof that a simple training procedure (the perceptron training rule - lecture 2) would converge if a solution to the problem existed.

Rumelhart and Zipser (1985) [31] give some interesting anecdotes and reminiscences from this era.

In 1969 enthusiasm for neural nets was dampened somewhat by the publication of Minsky and Papert's book 'Perceptrons' [27]. Here, the authors show that there is an interesting class of problems (those that are not linearly separable) that single layer perceptron nets cannot solve, and they held out little hope for the training of multi-layer systems that might deal successfully with some of these. Minsky had clearly had a change of heart since 1951, when he and Dean Edmonds had built one of the first network-based learning machines. The fundamental obstacle to be overcome is the so-called 'credit assignment problem': in a multilayer system, how much does each unit (especially one not in the output layer) contribute to the error the net has made in processing the current training vector? (This is the problem to which BP is addressed).

In spite of 'Perceptrons', much work continued in what was now an unfashionable area, living in the shadow of symbolic AI: Grossberg was laying the foundations for his Adaptive Resonance Theory (ART) [6, 11]. Fukushima was developing the cognitron [9]; Kohonen was investigating nets that used topological feature maps [22] (lecture 7), and Aleksander [2] was building hardware implementations of the nets based on the n-tuple technique of Bledsoe and Browning [5].

There have been several developments over the last few years that have led to a resurgence of interest in nets. Some of these factors are technical and show that Minsky and Papert had not said the last word on the subject, while others are more general; in the next section we look at both these strands of thought.

2.2 The neural net renaissance

In 1982 John Hopfield [19], then a physicist at Caltech, showed that a highly interconnected network of threshold logic units could be analyzed by considering it to be a physical dynamic system possessing an 'energy'. The process of associative recall, where the net is started in some initial random state and goes on to some stable final state, parallels the action of the system falling into a state of minimal energy.

This novel approach to the treatment of nets with feedback loops in their connection paths (recurrent nets), has proved very fruitful and has led to the involvement of the physics community, as the mathematics of these systems is very similar to that used in the Ising-spin model of magnetic phenomena in materials [3]. In fact something very close to the 'Hopfield model' had been introduced previously by Little , but here the emphasis was on this analogy with spin systems, rather than the energy-based description. Little [24] also made extensive use of a quantum mechanical formalism, and this may have contributed to the paper's lack of impact.

A similar breakthrough occurred in connection with non-recurrent (feedforward) nets, when it was shown that the credit assignment problem had an exact solution. The resulting algorithm, 'Back error propagation' or simply *Backpropagation* also has claim to multiple authorship, as noted by Grossberg in [12]. Thus it was discovered by Werbos [38] rediscovered by Parker [28], and discovered again and made popular by Rumelhart, Hinton and Williams [32].

Aside from these technical advances in analysis, there is also a sense in which neural networks are just one example of a wider class of systems that the physical sciences have started to investigate which include cellular automata [33], fractals [25], and chaotic phenomena [16]. Traditionally physics has sought to explain by supposing that there is some simple underlying global model, usually consisting of a differential equation, to which the real situation is supposed to approximate as a perturbation. Now some investigators are meeting the complexity of dynamical systems 'head on' as it were, and are supposing that any macroscopic ordering is an emergent property, arising from the action of a large number of simple stochastic elements acting in concert. Statistical physics represents the vanguard of this movement that now includes the areas noted above. Neural nets are also complex dynamical systems with emergent collective properties, as noted by Hopfield in the title of his original paper [19] and it is not surprising, in hindsight, that the solid-state statistical physicists have managed to apply their theories to the subject.

Now in order to investigate these new models, it is often necessary to resort to computer simulation, and the power of this method has obviously been greatly enhanced by advances in technology. Thus certain experiments would simply have been unthinkable fifteen years ago by the average worker, given the accessibility and power of the computing resources available.

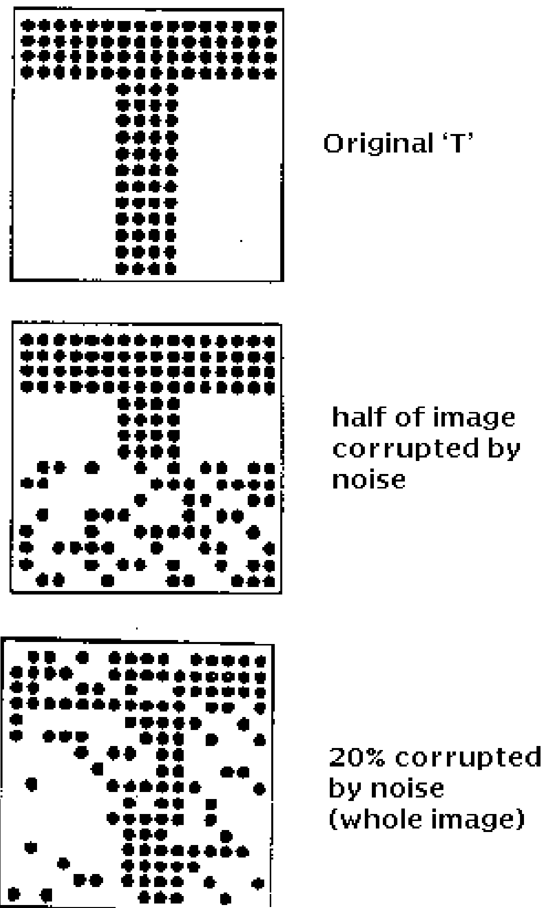
In summary therefore, we may say that there has, in recent years, been a marked increase in neural net research because of new mathematical insights, a conducive scientific *zeitgeist*, and enormous improvement in the ability to perform simulation.

3 Classifying neural net structures

On a first exposure to the neural net literature, it may seem that the whole area is simply a large collection of different architectures, node, types etc., which are somewhat *ad hoc* and which don't appear related in any way. It is the intention of this section to try and give a framework for classifying any network which enables it to be viewed as a special instance of, or as a composite of, only a handful of structures. All of these have been presented in previous lectures, we simply draw them together here for a comparative review. We start, however by reviewing the types of task that neural nets can perform.

3.1 Neural net tasks

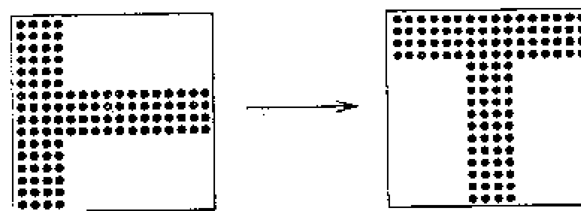
Recall the example of pattern recall with the 'T's and the examples in the Hopfield net demo (figure reproduced here for clarity).



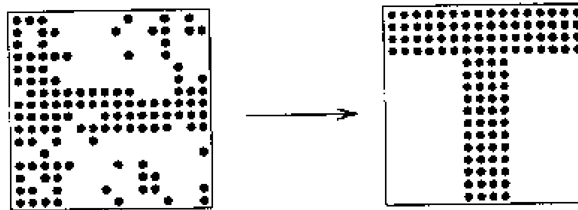
[slides of Ts]

The middle image is split into two halves: the top half or *key* is an uncorrupted portion of the original input vector; the rest of the input is random noise. If the net can retrieve the original training pattern from (b) then it is said to be acting as a *content addressable memory* or CAM. The lower image shows the original 'T' with the uniform addition of noise obtained by inverting each pattern element with probability 0.2. If the net can retrieve the original image then it is now behaving as a filter or noise extractor. Both of these are instances of *auto-associative recall* where a partial or noisy image is restored to some prototype.

The top image on the second slide shows a training vector pair, with input and output vectors on the left and right respectively.



(a) training vector pair



(b) recall from noisy input

[hetero-associative recall on ‘T’s]

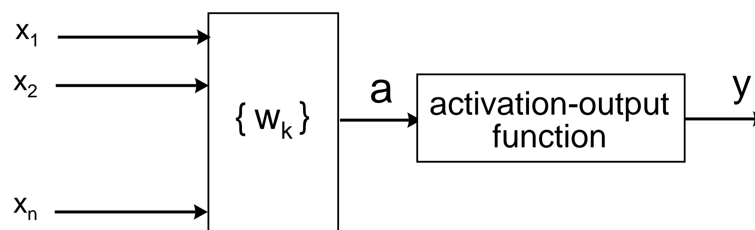
The lower image shows a noisy input recalling the original output vector. The split of the training pattern into two dissimilar halves, leads to the name *hetero-associative recall* for this process.

Suppose however, that in the last example we had unified the input and output patterns side by side into a single training vector. By keying on the left half of this and performing auto- association, we obtain a similar result (although not perhaps as noise resilient) to that obtained in the case of hetero- association. Thus conceptually both hetero- and auto-association can perform similar tasks; this point of view is useful later in comparing modes of learning.

As a final task, consider the hetero-associative case where the target or output pattern is much smaller than that on the input, and where it represents a highly compressed encoding of the input set (possibly with one node ‘on’, or several nodes ‘on’). In this case we talk of the net performing *pattern classification*, and introducing it in this way, classification can be seen as a special case of hetero-association.

3.2 A taxonomy of artificial neurons

It is useful to define the functional behaviour of all the unit types using the *activation-output* model introduced for the discussion of cubic nodes.



[activation-output model]

x_i is a set of n inputs, and ζ_k, a set of N internal parameters. The activation a is a function of the inputs and parameters, $a = a(\zeta_k, x_i)$. The output y is a function of the activation,

$y = y(a)$, defined by the activation-output relation. Splitting the node functionality up like this can assist the understanding of what 'ingredients' have been used in the node make-up.

We now describe some common node types and use these to draw up a list of criteria that may be used to classify them.

3.2.1 Nodes using a linear weighted sum of inputs

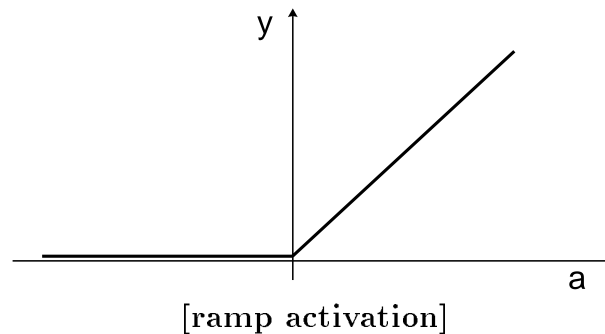
These are by far the most common. The parameters ζ_k now consist of a set of n *weights* w_i, one for each input, and a threshold parameter θ, that may be thought of as a weight associated with an input that is tied to '-1'. The activation a, is given by

$$a = \sum_{i=1}^{n} w_i x_i - \theta \tag{1}$$

The simplest possible activation-output a-o relation to use in conjunction with (1) is to put $y = a$. The resulting unit is found in the counterpropagation nets of Hecht-Nielson [14] Note that y takes on a continuous range of values and so networks of these units are computationally analogue.

Other possibilities for a-o include the threshold function used in TLUs which can therefore only use Boolean values. The other popular output law is based on the sigmoid function. Another function, used by Fukushima in his cognitron [18] is given by

$$y = \begin{cases} 0 & \text{if} \quad a < 0 \\ a & \text{if} \quad a \geq 0 \end{cases} \tag{2}$$



[ramp activation]

As well as using y directly as an analogue value, it is possible to interpret it as the probability that a '1' is emitted from the output in a stochastic, boolean valued node. This is used to good effect in the Reward Penalty algorithms [4] and the Boltzmann Machines [17].

The main feature of the hard-limiter and sigmoid functions is their nonlinearity, a feature which can be highly significant in the way that neural nets process information, and which Grossberg discusses at length in [12].

3.2.2 More complex nodes

It is possible to extend the simple activation relation (1) so that rather than just summing over terms linear in the inputs, we include higher order terms that are multilinear in these variables. This gives us the sigma-pi nodes. Any of the activation-output relations described above may be used with the sigma-pi activation.

Alternatively, we may define a node whose inputs and outputs are boolean, and which can perform *any* (perhaps stochastic) boolean function of its inputs (not necessarily linearly separable). The ζ_k are then the site-values at the corners of n-cube and we have the so-called cubic nodes.

3.2.3 Criteria for classification

In the light of the above examples, it is proposed that the following is a list of features that may act as dimensions in a 'node space' in the study of neural nets.

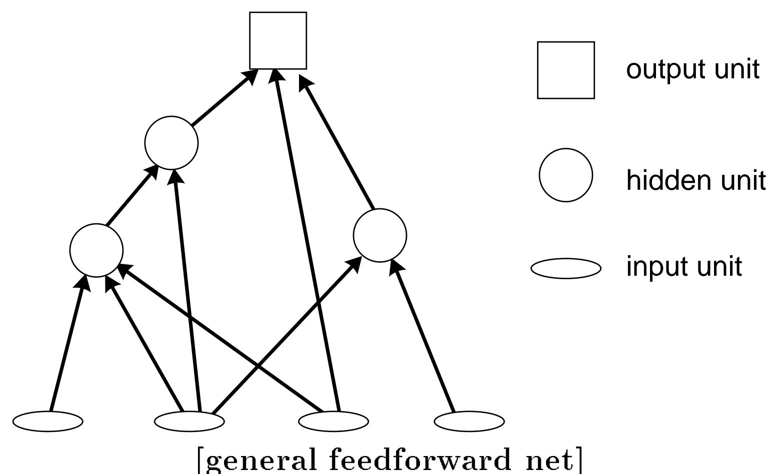
- The form of the function $a(\zeta_k)$ - e.g. linear, sigma-pi, cubic.
- The activation-output relation - linear, hard-limiter, or sigmoidal.
- The nature of the signals used to communicate between nodes - analogue or boolean.
- The dynamics of the node - deterministic or stochastic.

3.3 A taxonomy of net structures

The last section dealt with an analysis of the micro-structure of the net at node level; here we deal with the overall topology and dynamics. Quite generally, in connection with the latter, we usually recognize two phases of net operation: a *training phase*, when the node parameters ζ_k are being changed according to some learning rule, and a *testing phase* when these variables remain static and when we are particularly interested in the net's response to new, unseen patterns.

3.3.1 Feedforward nets

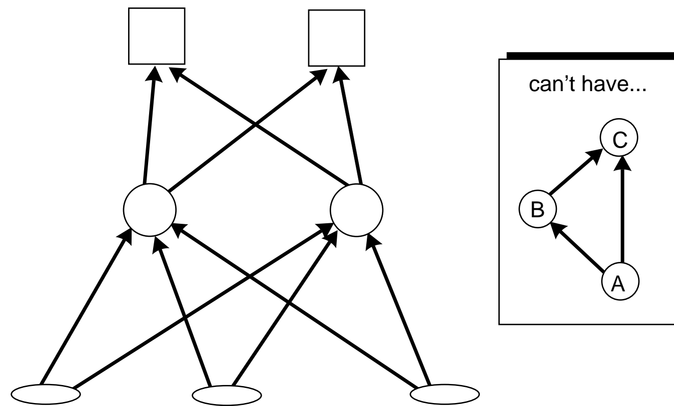
These are nets that contain no feedback loops in their inter-node connection paths. Thus, starting from any node in the net and tracing its fan-out, there will come a point after a finite number of steps when this process terminates at an *output node* which does not have a fan-out. Complementary to these are a set of *input nodes* or *input terminals* which have no fan-in; they have no functionality and are merely fan-out distribution points. The most general feedforward net is shown below



[general feedforward net]

The input and output nodes are accessible to the training environment and may therefore have signals directly imposed on them from outside. In between these two are a set of nodes referred to as *hidden* since they may not be accessed by the environment. It is the task of these units to develop an internal representation of the training set. In this context the hidden units are sometimes thought of as extracting *features*, or *encoding* the environment [10]. This is most apparent if there are fewer hidden than input nodes, and the net must perform some data compression. This is the basis for the operation of the encoder nets which were used in the simulation exercises.

One specialization of the general feedforward net is that where it is possible to distinguish a series of distinct layers. Formally this may be described in the following way: there does not exist a set of three units A, B and C, such that C receives input from A and B, and B receives input from A. The forbidden structure is shown below together with a typical 3-layer feedforward net.



[forbidden layered structure and a layered net]

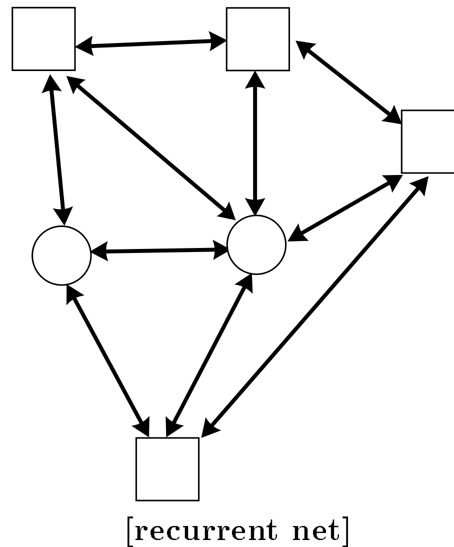
The dynamics of feedforward nets is straightforward. In the layered case, inputs are applied to the input nodes and subsequent layers evaluate their node outputs until the output layer is reached. In the more general case care has to be taken to ensure that all the inputs to a given node are valid before its evaluation. This whole process is referred to as a *forward pass.*

The perceptron rule and delta rule could be used to train feedforward nets with no hidden units. The backpropagation algorithm allowed training of hidden units in these nets.

3.3.2 Recurrent nets: topology

These are nets which have feedback loops in their inter-node connection paths. Thus there must exist at least one node such that, if the fan-out of that node is traced forward, then after a finite number of steps the node is revisited. In recurrent nets there is no distinction between input and output, rather there is a set of *visible units* which interface with the environment and, as in the feedforward case, a number of hidden units which do not. The way this interface operates is discussed below under 'dynamics'. The Hopfield net was an example of a recurrent net with no hidden units. The Boltzmann machines [17] (not dealt with on this course) allow hidden units

A typical recurrent net is shown below



[recurrent net]

The Hopfield net was a special case where there was total interconnectivity and all connections were symmetric.

3.3.3 Recurrent nets: dynamics

There are two fundamental modes of operation - *synchronous* or *parallel* update, and *asynchronous* update. In the former, all the nodes evaluate their new output together, and the net is 'clocked' globally. In the latter case, at each time step a node is selected at random to be updated. Both synchronous and asynchronous modes can be thought of as extreme instances of the more general case where there is a probability p that any node gets updated. For a comparison of net behaviour in these two modes see [10].

In the case of synchronous dynamics, and with deterministic nodes, the net is a *deterministic finite automaton* with a characteristic state space, where the state of the net is just the vector of unit outputs. These spaces may be partitioned into sets of states or *confluents* such that all states belong to only one confluent, and transitions are confined to pairs of states within each one. Examples of confluent structure were shown in the figure 'state diagrams for synchronous update' in lecture 5.

confluent structures

All confluents can be divided into two parts: a tree like fragment or *transient* and a *cycle*. In the left hand part of the figure, the cycle consists of a single state returning to itself - a single-state cycle or 1-cycle; in the right hand figure it includes several states, giving a multiple-cycle. Starting the net in a random state will usually mean beginning on a transient, and subsequent transitions will be made until a cycle is reached. Therefore this type of dynamics implies that we have to wait for the net to reach some kind of equilibrium.

Asynchronous operation implies the net is no longer deterministic, since nodes are selected for update at random. Therefore although we can define the state of the net at any time as before, the state structure is probabilistic and more complex than the one given above. (This leads to diagrams like the ones used with the Hopfield net). The concept of reaching equilibrium can be retained by the introduction of an 'energy function' associated with the net; this was one of the main insights given by Hopfield who showed that the process of coming to equilibrium may be thought of as one of energy minimisation. In this way memories could be thought of as states associated with stable energy minima, into which

random starting states would eventually fall. The energy is chosen so that each neuron update implies a decrease in its value, and so we imagine an energy 'landscape' where, even though the next state of the system is not known *a priori*, the transition is guaranteed to be 'downhill' in the energy space.

We now turn to the question of how the network interacts with the environment. In training, a vector will be applied to these nodes in such a way that their outputs take the same values as the corresponding vector components. If there are any hidden nodes, the net is allowed to reach equilibrium by allowing these to update, if there are none we proceed directly to the use of the learning rule where the weights are then changed.

Under test, there are (at least) two ways of inputting information. One is to clamp a subset of the visible units and keep the clamp on for the entire test, while the net reaches equilibrium. The clamp can be freely reselected at each trial from the entire collection of visible units. This corresponds to the use of the net as a content addressable memory (see 'node tasks' section) and is the mode used with the Boltzmann Machines. Alternatively we may initialize the state of the net from an input vector, and then let the net operate completely freely, with *all* nodes partaking in the update process: this is the preferred mode when using the Hopfield nets and corresponds to recall from noisy data. Note, however, that the Hopfield net may also be used as a CAM.

3.4 Competitive learning nets

These are a kind of hybrid, where a feedforward structure contains at least one layer with intra-layer recurrence (see diagram in lecture 7). Each node is connected to a large neighbourhood of surrounding nodes by negative or *inhibitory* weighted inputs, and to itself (and possible a small local neighbourhood) via positive or *excitatory* inputs. These lateral connections are usually fixed in magnitude and do not partake in the learning process. It is also connected to the previous layer by a set of inputs which have trainable weights of either sign. The point of the lateral recurrent links is to enhance an initial pattern of activity over the layer, resulting in the node which was most active being turned fully 'on', and the others being turned 'off'; hence the label 'competitive' or 'winner-take-all' being applied to these nets.

The object here is to encode groups of patterns in a 1-out-of- n code, where each class is associated with an active node in the winner-take-all layer. This mechanism, or some minor variant, is a key component in Grossberg's ART [6, 11] Fukushima's cognitron [9] and Kohonen's nets that develop topological feature maps [21].

3.5 Criteria for classification

What was done for nodes in the last section is now done for net structures, so that the following is proposed as a set of attributes for net classification.

- Basic net topology - Recurrent , competitive, or feedforward.
- Hidden units - present or absent.
- Dynamics - Synchronous or asynchronous.

3.6 A taxonomy of training algorithms

These fall into three main categories. These are described individually and are followed by a discussion of the relation between the environment and the net, and the resources required for training.

3.6.1 Preprogramming

In this scheme the weights are not learnt as such by a gradual, iterative process of training vector presentation and update, rather they are fixed according to some prescription that makes use of all the vectors at once; in a sense the net is not trained but programmed. The best known example of this occurs in the Hopfield model.

3.6.2 Supervised learning

This is usually performed with feedforward nets where training patterns are composed of two parts, an input vector and an output vector, associated with the input and output nodes respectively. A training cycle consists of the following steps. An input vector is presented at the inputs together with a set of desired responses, one for each node, at the output layer. A forward pass is done and the errors or discrepancies, between the desired and actual response for each node in the output layer, are found. These are then used to determine weight changes in the net according to the prevailing learning rule.

The term 'supervised' originates from the fact that the desired signals on individual output nodes are provided by an external 'teacher'. The best known examples of this technique occur in the backpropagation algorithm, the delta rule and perceptron rule.

A popular measure of the error E for a single training pattern, is the sum of square differences

$$E = \frac{1}{2} \sum_i (t_i - y_i)^2 \tag{3}$$

where t_i is the desired or target response on the ith unit, and $y_i$is that actually produced on the same unit. This is appealing because of its simplicity, but is somewhat *ad hoc* (why not use the fourth power of differences?) and Hopfield [20] has argued for the use of an alternative measure based on information theoretic ideas.

3.6.3 Unsupervised learning

This is usually found in the context of recurrent and competitive nets. There is no separation of the training set into input and output pairs. Typically a training cycle will consist of the following steps: a vector is applied to the visible nodes (or in the case of competitive learning, the input nodes); the net is allowed to reach equilibrium (or a 'winner' established); weight changes are made according to some prescription. It is the amalgamation of input-output pairs, and hence the disappearance of the external supervisor providing target outputs, that gives this scheme its name. This kind of learning is sometimes referred to as *self-organization*.

3.6.4 Computational resources and complexity

In [39] Williams makes the useful distinction between 'on-line' and 'off-line' learning. In the latter the net is trained not only with its own resources, but also with the aid of an auxiliary central computing facility. This orchestrates the learning process and may pass information to, and gather information from subsets of nodes in the network . This is clearly the case in the preprogramming style of learning used for Hopfield nets. It is also arguably the case in the backpropagation type algorithms, where errors and weight values are accumulated at a node from its fanout. In on-line training, the net only makes use of the facilities offered internally within each node, together with perhaps, a minimal number of control signals to initiate various stages in each learning cycle.

The on-line versus off-line distinction may therefore be reformulated as a local versus global one. Learning is local if processing is takes place principally in individual units, and if these make use of information immediately available both in space and time. This means that we are restricting operations to the use of a node's current inputs and output; storage of previously calculated values being prohibited. Learning is off- line if large amounts of data are continually being processed and redistributed around the net along pathways that do not form part of its natural connectivity.

Conceptually it may be possible to reformulate an off-line algorithm as an on-line one, if the boundary between the environment and the net is redrawn, so that resources previously within the 'net' are now thought of as within a more supervisory environment. In spite of the informality of these distinctions, it is believed useful to include them in the general net taxonomy.

3.7 Final remarks

Clearly not all nets will fall neatly into the categories described here - some nets will contain mixtures of features described here - but it is believed that some such taxonomy is useful in approaching a field which now has a large proliferation of architectures, structures and algorithms. One facet which has not been dealt with here is the relation between the operation of neural nets and conventional pattern classification and recognition techniques. Some of these links are established by Lippmann [23] who also gives a review of some specific neural net types.

4 Neural nets in different disciplines

Neural nets have attracted attention from workers in many, apparently disparate, fields. It is ironic that the interdisciplinary fusion that was cybernetics should have fallen into disrepute in the West (the word is still fashionable in the Eastern bloc). Much of the work done in cybernetics now goes under the title of 'General Systems Theory'. Neural nets would be subsumed under its general remit and the interdisciplinary links required for it to flourish would have been forged. There are other, related areas which would now also fall under the cybernetics umbrella, and from which the study of neural nets could benefit. These include the use Genetic Algorithms (GAs), the study of cellular automata, robotics, adaptive control and mainstream AI. Many scientists now recognise that there is a common thread in all this research and the term 'artificial life' has been coined to embrace these areas collectively. The key question appears to be "How can assemblies of simple system elements exhibit self-organising behaviour patterns and adapt to their environment in ways that appear intelligent?".

Returning to the confines of neural networks, their study can be approached from several different points of view leading to different ideas as to their utility and the goals of the research.

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. Many biologists are sceptical about the ultimate power of some of the more impoverished models of neurons like TLUs and insist that more detail is necessary to explain the brain's function; only time will tell.

On the other hand, engineers and computer scientists see neural nets as one style of *parallel distributed computing* which may usefully be recruited for solving complex problems in pattern recognition and classification, associative memory, and function optimisation. The hope is that they may be more successful in dealing with real-world situations than the

conventional algorithmic techniques that have predominated in machine intelligence. They do not concern themselves with biological realism and are perhaps culpable of sometimes ignoring relevant architectural cues from natural systems.

Physicists and mathematicians are drawn to the study of networks from an interest in non-linear dynamical systems, statistical mechanics and automata theory. They often treat recurrent nets by (like Hopfield nets) by taking the limit as the number of neurons tends to infinity and dealing with the resulting network as a thermodynamic system which avails itself of all the machinery of statistical mechanics. There is contact to be made here with some of the concepts in adaptive control theory and the use of an 'energy' function may be traced back to Lyapunov; hence they are often referred to as Lyapunov functions.

Some of the threads running through much of this work are the ability of neural nets to learn and self-organise, to generalise from training data, and to process information in other ways normally thought of as intelligent.

References

[1] I. Aleksander. Whatever happened to cybernetics. Technical Report N/S/103, Dept. Electrical Engineering, Brunel University, 1980.

[2] I. Aleksander and T.J. Stonham. Guide to pattern recognistion using random-access memories. *Computers and digital techniques*, 2:29 – 40, 1979.

[3] D.J. Amit and H. Gutfreund. Spin-glass models of neural networks. *Physical Review A*, 32:1007 – 1018, 1985.

[4] A.G. Barto and M.I. Jordan. Gradient following without backpropagation in layered networks. In *1st Int. Conference Neural Nets, San Diego*, volume 2. 1987. (I have this).

[5] W.W. Bledsoe and I. Browning. Pattern recognition and reading by machines. In *Proceedings of the Eastern Joint Computer Conference*, pages 225 – 232. 1959.

[6] G. Carpenter and S. Grossberg. A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Processing*, 37:54 – 115, 1987.

[7] J. Casti. *Connectivity, Complexity and Catastrophe in Largescale Systems.* Wiley, 1979.

[8] H.L. Dreyfus. *What Computers Can't Do - The Limits of Artificial Intelligence.* Harper and Row, 1979.

[9] K. Fukushima. Cognitron: a self-organizing multilayered neural network. *Biological Cybernetics*, 20:121 – 136, 1975.

[10] R.O. Grondin, W. Porod, C.M. Loeffler, and D.K. Ferry. Synchronous and asynchronous systems of threshold elements. *Biological Cybernetics*, 49:1 – 7, 1983.

[11] S. Grossberg. Competitive learning: from interactive activation to adaptive resonance. *Cognitive Science*, 11:23 – 63, 1987.

[12] S. Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks*, 1:17 – 61, 1988.

[13] D.O. Hebb. *The Organization of behaviour.* John Wiley, 1949.

[14] R. Hecht-Nielson. Counterpropagation networks. In *1st Int. Conference Neural Nets, San Diego*, volume 2. 1987.
(I have this).

[15] S.J. Heims. *John von Neumann and Norbert Weiner - From Mathematics to the Technologies of Life and Death.* Academic Press, 1982.

[16] A. Hilger. *Universality in Chaos.* ?, Bristol.

[17] G.E. Hinton, T.J. Sejnowski, and D. Ackley. Boltzmann machines: Constraint satisfaction networks that learn. Technical Report CMU-CS-84-119, Carnegie Mellon University, 1984.

[18] A. Hodges. *Alan Turing - The Enigma of Intelligence.* Counterpoint (Unwin), 1985.

[19] J.J. Hopfield. Neural networks and physical systems with emergent collective computational properties. *Proceedings of the National Academy of Sciences of the USA*, 79:2554 – 2588, 1982. Hopfield has another, related, model which uses continuous outputs. Beware when reading the literature which model is being discussed.

[20] J.J. Hopfield. Learning algorithms and probability distributions in feed-forward and feedback networks. *Proceedings of the National Academy of Sciences of the USA*, 84:8429 – 8433, 1987.

[21] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59 – 69, 1982.

[22] T. Kohonen. *Self-organization and associative memory.* Springer Verlag, 1984.

[23] R.P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4 – 22, 1987.

[24] W.A. Little. The existence of persistent states in the brain. *Mathematical Biosciences*, 19:101 – 120, 1974.

[25] B.B. Mandelbrot. *The Fractal Geometry of Nature.* Freeman, 1977.

[26] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7:115 – 133, 1943.

[27] M. Minsky and S. Papert. *Perceptrons.* MIT Press, 1969.

[28] D.B. Parker. Learning-logic. Technical Report 581-64, Office of Technology Licensing, Stanford University, 1982.

[29] R. Raj. Foundations and grand challenges of artificial intelligence. *AI Magazine*, 9:9 – 21, 1988.

[30] F. Rosenblatt. *Principles of Neurodynamics.* Spartan Books, 1962.

[31] D. Rumelhart and D. Zipser. Feature discovery by competitive learning. *Cognitive Science*, 9:75 – 112, 1985.

[32] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 – 536, 1986.

[33] Several authors. Cellular automata - proceedings of an interdisciplinary workshop, los alamos. In *Physica, vol. 10D, p. (special volume)*. van Nostrand ?, 1983.

[34] P Smolensky. On the proper treatment of connectionism. *Behavioural and Brain Sciences*, 11:1 – 74, 1988.

[35] John von Neumann. First draft of a report on the edvac. In W. Aspray and A. Burks, editors, *Papers of John von Neumann on Computing and Computer Theory, vol 12 in the Charles Babbage Institute Reprint Series for the History of Computing*. MIT Press, 1987.

[36] John von Neumann. The general and logical theory of automata. In W. Aspray and A. Burks, editors, *Papers of John von Neumann on Computing and Computer Theory, vol 12 in the Charles Babbage Institute Reprint Series for the History of Computing*. MIT Press, 1987.

[37] D.L. Waltz and J.B. Pollack. Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9:51 – 74, 1985.

[38] P. Werbos. *Beyond regression: New tools for prediction and analysis in the behavioural sciences*. PhD thesis, Harvard University, Cambridge, MA., 1974.

[39] R.J. Williams. Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3, Northeastern University Boston, 1987.