Chapter 5

Decisions: Neural Nets(Old Style)

In the second chapter I suggested ways of turning an image or a part of an image, into a vector of numbers. In the last chapter, I showed, *inter alia*, how to model a collection of points in $\mathbb{R}^n$ by gaussians and mixtures of gaussians. If you have two or more categories of point (paint them different colours) in $\mathbb{R}^n$, and if you fit a gaussian or mixture of gaussians to each category, you can use the decision process (also described in the last chapter) to decide, for any new point, to which category it probably belongs.

It should be clear that in modelling the set of points belonging to one category by gaussians (or indeed any other family of distributions) we are making some assumptions about the nature of the process responsible for producing the data. The assumptions implicit in the gaussian mixture *pdf* are very modest and amount to supposing only that a large number of small and independent factors are producing unpredictable fluctuations about each of a small number of 'ideal' or 'template' stereotypes described by the measuring process. This is, frequently, not unreasonable: if we are reading printed text, we can suppose that there are several ideal shapes for a letter /A/, depending on whether it is italic, in Roman or sans-serif font, and that in addition there are small wobbles at the pixel level caused by quantisation and perhaps the noise of scanning. There should be as many gaussians for each letter as there are distinct stereotypes, and each gaussian should describe the perturbations from this ideal. So the approach has some attractions. Moreover, it may be shown that any *pdf* may be approximated arbitrarily closely by a mixture of gaussians, so even if the production process is more complex than the simple model suggested for characters, it is still possible to feel that the model is defensible.

If we take two clusters of points, each described by a gaussian, there is, for any choice of costs, a decision hypersurface separating the two regions of $\mathbb{R}^n$ containing the data, as in the figure. This hypersurface is defined by a linear combination of quadratics and hence is itself the zero of some quadratic function. In the particular case when both clusters have the same covariance matrix, this reduces to a hyperplane. If the covariance matrices are not very different, then a hyperplane between the two regions will still be a fair approximation in the region between the two clusters, which is usually the region we care about. And you can do the sums faster with an affine hyperplane, so why not use hyperplanes to implement decision boundaries? Also, we don't usually have any good grounds for believing the clusters are gaussian anyway, and unless there's a whole lot of data, our estimates of the covariance matrices and centres are probably shaky, so the resulting decision boundary is fairly tentative, and approximating it with a hyperplane is quick and easy. And for more complicated regions, why not use piecewise affine decision boundaries? Add to this the proposition that neurons in the brain implement affine subspaces as decision boundaries, and the romance of neural nets is born.

In this chapter, I shall first outline the history and some of the romance of neural nets. I shall explain the Perceptron convergence algorithm, which tells you how to train a single neuron, and explain how it was modified to deal with networks of more than one neuron, back in the dawn of neural net research. Then I shall discuss the hiatus in Neural Net research caused by, or at least attributed to, Marvin Minsky, and the rebirth of Neural Nets. I shall explain layered nets and the Back-

Propagation algorithm. I shall discuss the menagerie of other Articial Neural Nets (ANNs for short) and indicate where they fit into Pattern Recognition methods, and finally I shall make some comparisons with statistical methods of doing pattern Recognition.

There was an intererESTing exchange on the net in 1990 which concerned the relative merits of statistical and neural net (NN) models. Part of it went as follows:

' ... I think NNs are more accessible because the mathematics is so straightforward, and the methods work pretty well even if you don't know what you're doing (as opposed to many statistical techniques that require some expertise to use correctly)' ......

However, it seems just about no one has really attempted a one-to-one sort of comparison using traditional pattern recognition benchmarks. Just about everything I hear and read is anecdotal. Would it be fair to say that "neural nets" are more accessible, simply because there is such a plethora of 'sexy' user-friendly packages for sale? Or is back-prop (for example) truly a more flexible and widely-applicable algorithm than other statistical methods with uglier-sounding names? If not, it seems to me that most connectionists should be having a bit of a mid-life crisis about now.'

From Usenet News, comp.ai.neural-net, August 1990.

This may be a trifle naive, but it has a refreshing honesty and clarity. The use of packages by people who don't know what they're doing is somewhat worrying, if you don't know what you're doing, you probably shouldn't be doing it, but Statistics has had to put up with Social Scientists (so called) doing frightful things with SPSS and other statistical packages for a long time now. And the request for some convincing arguments in favour of Neural Nets is entirely reasonable and to be commended. The lack of straight and definitive answers quite properly concerned the enquirer.

The question of whether neural nets are the answer to the question of how brains work, the best known way of doing Artificial Intelligence or just the current fad to be exploited by the cynical as a new form of intellectual snake oil, merits serious investigation. The writers tend to be partisan and the evidence confusing. We shall investigate the need for a mid-life crisis in this chapter.

5.1 History: the good old days

5.1.1 The Dawn of Neural Nets

In the 'classical' theory of Neural Nets, which we may trace to Rosenblatt, Block, Widrow and Nilsson, not forgetting the early work of Pitts and McCulloch, there are two strands. One is the idea of piecewise affine (or piecewise linear as they used to say in the bad old days) hypersurfaces as discriminating boundaries, and the other is the belief that this is how brains might do the job of classifying data. I have already discussed the former in the case of the archetype pattern recognition problem of telling the guys from the gals.

The idea that a neuron might be a *threshold logic unit*, that is to say a unit where some weighted sum of inputs was compared with a threshold and either output a 1 or -1, according to whether the weighted sum exceeded the threshold or not, goes back to the observation that sometimes neurons fired, in which case a spike of voltage difference between inside and outside the neuron travelled along it, from input end to output, afferent dendrites to efferent dendrites, until it came to another neuron, at which point the following neuron, given a sufficient amount of stimulus from the first and possibly other neurons, might also fire. So the firing of a neuron came to be seen as an all or none affair, mediated by the transmission coefficients at the *synapses*, the places where one neuron impinged on another. This made it a *weighted majority gate*, in an alternative terminology, with each weight representing some state of a synapse. The *Hebb doctrine*, that memory was stored in transmitter substance density and possibly neuronal geometry, also contributed to the view of neurons as adaptive threshold logic units. This is not a view

Figure 5.1: A sketch of a real, live neuron. Well, sort of.

which has survived, but it had a substantial effect on the thrust of the development of algorithms in the classical period from about 1940 to the sixties.

The sketch of a 'typical' neuron, as in *Fig.5.1* is a biological version of a big brother of *Fig.1.6*, with more inputs. Engineers tend to prefer straight lines and right angles to the wiggly curves of biology.

Rosenblatt, in 1961, had seen the *elementary perceptron*, just a single one of these units, as a model for a single neuron, and conceded without shame that the critical issue of how more units could combine to give larger perceptrons was still open. He used a particular adaptation rule called the *alpha rule* for training his elementary perceptron to output either $+1$ or -1, for use as a Pattern Classifier. It was well known that if the inputs were taken from the so called *XOR* data set, which had points at (0,0), (0,1), (1,0) and (1,1) where the first and last were of category 0 and the middle two of category 1 (and the name being chosen for reasons obvious to anyone with any familiarity with logic) then the

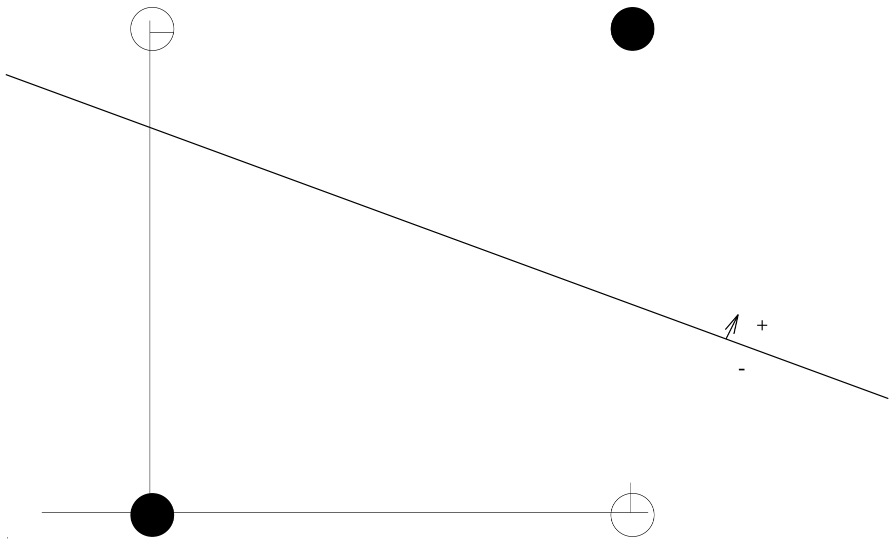

Figure 5.2: XOR: a classification problem beyond the capacity of the single neuron alpha-perceptron.

elementary alpha perceptron could not classify the data correctly. This is rather simple geometry. The elementary perceptron with input the coordinates of points in the plane was simply an oriented line which fired if the input data was on one side and didn't if it was on the other. And the XOR data set was not linearly separable.

Nilsson in 1965 had shown how what he called a 'committee net' of three neurons, all fed the same input, could all feed output to a final neuron, the final neuron seeing the preceding layer as voting on the correct classification of the datum. By simply taking the majority verdict, easily accomplished by a suitable choice of weights, the final neuron could yield the correct classification. We shall return to this shortly.

5.1.2 The death of Neural Nets

The late sixties saw the publication of Minsky and Papert's *Perceptrons*. This casually identified all perceptrons in the sense of Rosenblatt with the elementary alpha-perceptron, preceded by some local process which 'extracted features' from an image by looking at only some of the pixels within a region, and returned a vector of binary results which was then the input to a single model neuron. This did less than justice to the possibilities for (a) looking at the whole image and (b) using many units and hence having piecewise affine functions instead of just affine functions. By pointing out the limitations of this rather skeletal abstraction in a, perhaps, rather confusing manner, Minsky and Papert managed to convince most readers that there were easy problems which their brains could solve which the perceptron model family could not. The cover of the book showed two diagrams, one a double spiral and the other a single spiral, and explained that it was hard for a perceptron to tell the difference. It was pretty hard for the reader, too, and involved some tracing of paths with fingers, but this was overlooked. Similarly, it was shown that combining local viewpoints could never guarantee to tell if a pixel set is connected or not. Rather like the proof that no (finite state) computer can add any two numbers, which it somewhat resembles, this is a result of limited practical utility.[1] The whole range of all possible perceptrons, in the sense of Rosenblatt, would encompass what are now called recurrent nets as well as time delay nets and indeed just about all the extant neural nets. Minsky clobbered only a limited class, but he used Mathematics, and so hardly anybody troubled to read the fine print. What he was in fact trying to tackle was the issue of when local information about the geometry of an object could be condensed by a linear or affine map and then combined with other such local compressions to give global properties of the object. There is some reason to doubt that this is a good problem. There is even more reason

[1] It is true that no computer can add any two numbers. The point is that there are always numbers too big to be represented in the computer, no matter how big the computer is. Since we can make computers which can add numbers in the range we usually care about, this is not quite so devastating as it sounds to the innocent and uncritical.

to doubt if this problem has a lot to do with how brains work. Still, it is no dafter than much Mathematical research and was harmless in all respects except its effect on Neural Net work.

The moral to be drawn from this would seem to be that most people don't read books very carefully, especially if they contain mathematics. It might have helped if Minsky hadn't had such a high reputation, or if he had taken more care to explain that the limitations of the systems of the sort he was analysing were not pertinent to the entire model class. Minsky took it that the input to a perceptron was a discrete retina always, and that the values were binary only, and that it was impractical to look at other than local regions or to use other than affine functions. Thus the limitations of the perceptrons Minsky analysed were fairly severe from the beginning, and it was unfortunate that sloppy reading left the many with a vague notion that *all* perceptrons were being disposed of. There is a suspicion that the critical tone of the book was the result of a disenchanted love affair: the AI community had imagined that all their problems would be solved by just bunging them into a huge neural net and waiting for the computer to grind to a conclusion. Ah well, the sixties were like that. If MIT had Perceptrons, the hippies had flower power. Both sobered up eventually. MIT AI Lab did it first, but at the expense, perhaps, of a bad case of the snits.

The belief that Minsky had shown that Perceptrons couldn't do anything non-trivial, came at about the same time that disenchantment was beginning to set in among funding officers. It is folk-lore that Rosenblatt had spent around Two Million US Dollars by the time of Minsky's book and had little to show for it, at least by the standards of the military[2]. So it is possible that Minsky merely drove the nail a little further into the coffin. Anyway, *something* around this time more or less stopped research on piecewise affine systems dead in its tracks. But you can't keep a good idea down,[3] and eventually by changing the name to Artificial Neural Nets (ANNs) and later MultiLayer Perceptrons (MLPs) (when the variety of these fauna started to proliferate),

[2]That is, he hadn't killed anyone yet.

[3]Or a particularly half baked one either, alas. Astrology has been going much longer than Science, which may yet turn out to be a passing fad.

they came back. They are now the staple of snake oil merchants everyhere, alas, and we need another Marvin Minsky to come again and tell us what they can't do. But this time, do it right.

For Minsky's comments on the dispute, see *Whence Cybernetics*, **Connections**, The Newsletter of the IEEE Neural Networks Council, Vol.3. No.3., p3., September 1993. And also P4. It should be remarked that some of the views expressed show an extensive innocence.

5.1.3 The Rebirth of Neural Nets

The belief of Minsky in the late seventies appeared to be that what was needed was an effective way of training multilayer perceptrons. The quick fix committee nets of Nilsson were ignored for reasons that seem to be accidental. Training a single neuron by the method of Rosenblatt (and Widrow, and others) didn't appear to generalise in a way that people thought right and natural. Certainly, training the committee net was found experimentally not to always work, but the *ad hoc* fix of adding more units usually managed to give success. Just why was not entirely clear. And then came the the idea of following the linear summation by a smooth [4] approximation to the step function that had been used up to this point. This allowed one to use a little elementary calculus, together with a few minor fudges, to train nets of arbitrary complexity. Actually, what it really did was to give a rationale for doing what was being done anyway, but such is the power of myth that a belief was born that the new neural nets were the answer to the prayers that the Artificial Intelligentsia muttered to their heathen and doubtless robotic gods. The triumph of hope over experience seems to be an enduring feature of human existence.

The *Back-Propagation* algorithm then allows one to train multilayer perceptrons, the species of Neural Net to which you were introduced in the first chapter. They don't always converge to a solution, but it

[4] 'Smooth' is Mathematician's talk for differentiable as many times as looks like being a good idea

is found experimentally that adding a few extra units often works, just as with committee nets. This is not a coincidence. It is also found that the net can learn only rather simple patterns in a reasonable time, and that huge times are needed for the case where the data set is at all complicated. I shall discuss the algorithm a little later, and explain why it behaves as it does.

At the same time, the neurophysiologists were finding out what was being done and were less than enthusiastic about MLPs as models of anything inside anybody's head. The evidence against the core idea of neurons as adaptive weighted majority gates (which fired provided that the weighted sum of inputs exceeded some threshold), has been accumulating. While some may possibly work in this way, there seems to be much more than that going on in brains.

Other types of neural net were investigated around this time, and a number have survived. I shall deal with them rather briefly in later sections. Among them, the Kohonen net, the Adaptive Resonance net of Grossberg, the Probabilistic Neural Nets of Specht, the Hopfield nets and various other 'architectures' such as the *Neocognitron* are going to get some attention. In surveying the plethora of Neural Nets at the present time, there is a strong sense of having entered a menagerie, and moreover one on which no taxonomist has been consulted. I shall try to reduce the sense of anomie and focus on what kinds of things these nets accomplish, rather than on drawing wiring diagrams, which appear to be the main conceptual tool offered to the perplexed student.

My collaborator of many years, Dr. Chris deSilva, has written an elegant account of certain of these and has kindly given me permission to include a substantial article he has written explaining how some of them work.

5.1.4 The End of History

The above history constitutes a somewhat brief and stark introduction to the subject of ANNs or Neural Networks; it is, however, lacking in that corroborative detail required to add artistic verisimillitude to an otherwise bald and unconvincing narrative. The remainder of the chapter will explain to those who can use the packages, just what they are doing when they use them.

5.2 Training the Perceptron

It is simplest to return to the two dimensional case of Fred and Gladys from chapter one in order to find out what is happening with the single unit alpha-perceptron. I hear alarm bells ringing; they warn me that doing the solution in the case of dimension two with only two categories, and indeed only two data points, will convince the reader he is not getting his money's worth. That there has to be more profundity than this, or why do all the other books have so much more algebra? To conceal the lack of insight of the author, dear reader. Trust me.

Recall *Fig. 1.7*, which I have recycled as *Fig.5.3* and the more or less obligatory network diagram *Fig.1.6*, which I have recycled as *Fig.5.4*.

Now let us initialise the line $ax+by+c = 0$, or if you prefer the numbers a, b, c at random inside some interval. I shall suppose that none of them are zero, as seems reasonable, and if you wish to choose to have each of the numbers between, say, -1 and $+1$, I should not mind at all, although I have no particular preferences.

If you imagine that line A of *Fig.5.3* is the result of this intialisation, then it will give us a concrete picture of the start of operations.

It is worth noting that in dimension n, I should have to choose $n + 1$ numbers to specify a hyperplane. We have not yet given a name to the initial line or hyperplane, so I shall remedy this immediately by

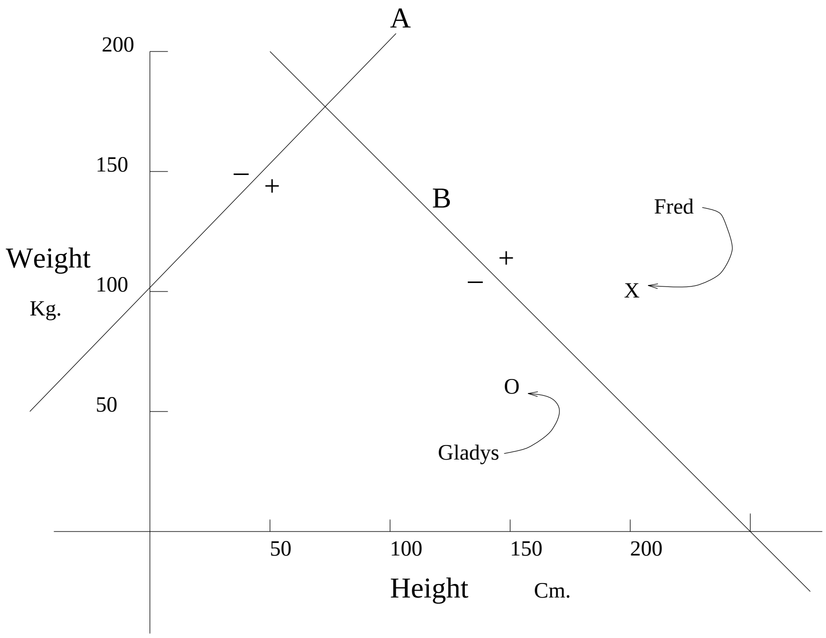

Figure 5.3: Two different states, A and B, of a neural net.

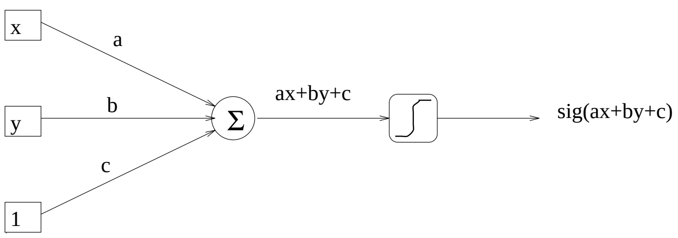

Figure 5.4: The neural net of the above diagram, in general.

calling it $\mathbf{a}$. It is not a very romantic name for anything, not even a hyperplane, but it is at least short.

Next I select a point from the data set, also at random. In our application this means picking either Fred or Gladys. In general there would be, one might hope, some rather larger pool of talent. Since I don't wish to prejudice the discussion by leading the reader to believe that it matters which one I pick, I shall call my randomly selected data point $\mathbf{x}$. If you want to ask awkward questions about randomness, I evade them by requiring only that your pseudo-random number generator be reasonably uniform in its selections and not, for example, too flagrantly sexist in the case of Fred and Gladys.

Next I shall work out on which side of the line $\mathbf{a}$ the point $\mathbf{x}$ lies. The simplest algebraic way of specifying this is to *augment* the point $\mathbf{x}$ to be a point $\hat{\mathbf{x}}$ in $\mathbb{R}^{n+1}$ which has all the components of the vector $\mathbf{x}$ with a 1 inserted in the $(n+1)^{th}$ place. If we do this simple trick, then the summation of the inputs x and y weighted by a and b with c added on, is simply expressed by the dot product $\mathbf{a} \cdot \hat{\mathbf{x}}$. Now I threshold this value to take the sign of it, $+1$ if $\mathbf{a} \cdot \hat{\mathbf{x}} \geq 0$, -1 if $\mathbf{a} \cdot \hat{\mathbf{x}} < 0$. That is to say, I obtain $sgn(\mathbf{a} \cdot \hat{\mathbf{x}})$. The reader will note that although he is probably thinking of $\mathbf{a}$ as a line and $\hat{\mathbf{x}}$ as a point to which a 1 has been appended, there is no reference to the dimension in these expressions, so they make sense for points and hyperplanes in $\mathbb{R}^n$ for any n whatever.

Next I want to think about the category or class of the datum, $\mathbf{x}$. This is either $+1$ or -1 in my system of representing such things; It is a value which belongs with the point and so I shall define the map from the data set of points $\mathcal{D}$ to ± 1

$$c : \mathcal{D} \longrightarrow S^0$$

which assigns to each point $\mathbf{x}$ its class.[5]

[5] S^0 is the zero dimensional sphere for those who don't know; the n-dimensional sphere is the equator of the $n+1$-dimensional sphere and the 1-dimensional sphere is a circle. The equator of a circle consists, of course, of the two points $\{\pm 1\}$. Now you know.

Note that this makes it fairly clear that the point of this business is to produce a map of the form $sgn(\mathbf{b} \cdot -)$ for some neural state $\mathbf{b}$, which takes the point $\mathbf{x}$ to the number $sgn(\mathbf{b} \cdot \hat{\mathbf{x}}), \pm 1$. This map will be defined for any $\mathbf{x}$ in $\mathbb{R}^n$, so it will tell us the class, 1 or -1 of every point $\mathbf{x}$. It is of course essential that this map agrees with c on the data set and extends it to all other points in $\mathbb{R}^n$. In essence, this is not different in principle from my giving you a dozen x and y values and asking you to draw a smooth curve through them all; you are fitting a function to a set of data. Whereas you usually want to fit a smooth function, perhaps a polynomial, here you have to fit a function which takes values in S^0, and which is given on points in $\mathbb{R}^n$, but you are still function fitting. That is, you have a finite function given you, and you want to extend it to the whole space. The curve fitting aspects may be emphasised by returning briefly to the diagram $fig.$1.11 of chapter one. Instead of fitting a smooth sigmoid as shown there, we would be fitting a step function, but the idea is the same. In dimension two, we should be fitting a step that runs right across the plane. Clearly we can focus on the dividing line, or more generally the separating hyperplane.

Now either $c(\mathbf{x}) = sgn(\mathbf{a} \cdot \hat{\mathbf{x}})$ or it doesn't. If it does, then we may say that the line $\mathbf{a}$ *correctly classifies* $\mathbf{x}$, or that $\mathbf{x}$ is on the right side of the hyperplane $\mathbf{a}$. Under these conditions we do absolutely nothing about changing the line $\mathbf{a}$.

The other possibility is that $c(\hat{\mathbf{x}}) = -sgn(\mathbf{a} \cdot \hat{\mathbf{x}})$. In this case we need to beat $\mathbf{a}$ around a bit and change its state. It is a naughty neuron which is getting the wrong answer, so it should be smacked. None of this post modernist Spockery here, no praising the neuron for being right and talking gently to it when it is in error; if it is right we ignore it and if wrong we kick it.

To see how to kick it is an interesting exercise in data representation. The AI fraternity, who tell us that the representation makes a big difference, are right.

The idea is to observe that there are a lot of possible $\mathbf{a}$ lines, and there has to be a rule for operating on each such $\mathbf{a}$ and turning it into a more compliant, better behaved $\mathbf{a}$. So we look at the space of all such $\mathbf{a}$. In the case of the recalcitrant line, the space is the space of three numbers a, b, c. The actual line $\mathbf{a}$ is a point in this space. This may be confusing at first, but it is easy enought to see that $2x + 3y + 4 = 0$ can be represented in the space of all such lines as the vector $\begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix}$.

Now the line has been turned into a point in this space of all lines in the plane, and the datum $\hat{\mathbf{x}}$ also defines a point in this space. And the latter point gives the right dot product with half the points in the space, and the wrong dot product with the other half. And for the case when the dot product is zero, the turn around occurs. In this dual space of lines, otherwise called the *weight space*, $\{\mathbf{a} : \hat{\mathbf{x}} \cdot \mathbf{a} = 0\}$ is a plane through the origin. It divides the space of lines into two parts, those that are right and those that are wrong. So it is an error to think of the augmented data point as being a vector in the space of all lines. It is better to think of it as a plane through the origin in that space. I draw a rather unconvincing picture of a bit of it in *Fig. 5.5.*

Now given that the line $\mathbf{a}$ is on the wrong side of the plane through the origin normal to the datum point $\hat{\mathbf{x}}$ (and if you can take that without blinking you are either on top of this or not with us at all), the obvious thing to do is to reflect it through the plane until it is on the right side. And this is the Perceptron convergence algorithm. Moreover, this way of thinking about it, in the weight space, gives some insight into why it works, which beats memorising the recipe.

To do the sums is straightforward and requires only the most elementary linear algebra. Suppose $c(\mathbf{x}) = 1$. Since the neural state $\mathbf{a}$ is wrong about the point, we must have that $\hat{\mathbf{x}} \cdot \mathbf{a} < 0$. The projection of $\mathbf{a}$ on $\hat{\mathbf{x}}$ is pointing in the direction opposite to $\hat{\mathbf{x}}$ and is the vector $\frac{\mathbf{a} \cdot \hat{\mathbf{x}}}{\hat{\mathbf{x}} \cdot \hat{\mathbf{x}}} \hat{\mathbf{x}}$. We need to subtract twice this vector from $\mathbf{a}$ if $\mathbf{a}$ is to be reflected through the plane through the origin normal to $\hat{\mathbf{x}}$ and made into a better and purer line which is right about the data point $\mathbf{x}$.

If $c(\hat{\mathbf{x}}) = -1$, then $\hat{\mathbf{x}} \cdot \mathbf{a} > 0$ since the line is wrong about the point,

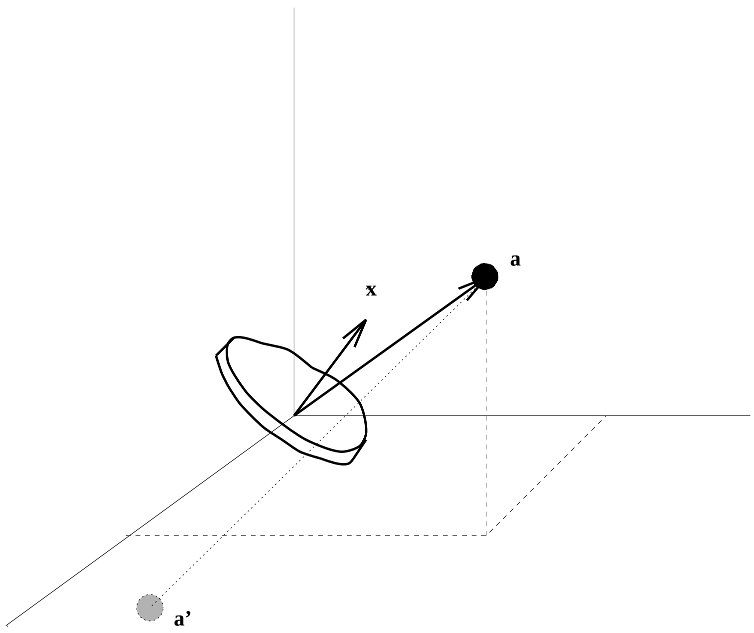

Figure 5.5: Into the dual space.

and so the projection of $\mathbf{a}$ on $\hat{\mathbf{x}}$ is positive and shouldn't be. Both are on the same side of the normal plane through the origin. So again we subtract the vector

$$2\frac{\mathbf{a} \cdot \hat{\mathbf{x}}}{\hat{\mathbf{x}} \cdot \hat{\mathbf{x}}}\hat{\mathbf{x}}$$

from $\mathbf{a}$. The Perceptron Training rule is then:

The Perceptron Training Rule

From a random initial state of the unit, $\mathbf{a}$ and with a randomly selected data point, $\mathbf{x}$, of category given by $c(\mathbf{x})$, augmented to $\hat{\mathbf{x}}$ by appending a 1 in the $n+1^{th}$ place,

1. evaluate $\mathbf{a} \cdot \hat{\mathbf{x}}$. If $sgn(\mathbf{a} \cdot \hat{\mathbf{x}}) = c(\mathbf{x})$ select a new data point at random and repeat from the last value of $\mathbf{a}$.

2. If $sgn(\mathbf{a} \cdot \hat{\mathbf{x}}) \neq c(\mathbf{x})$, alter the state $\mathbf{a}$ to $\mathbf{a}'$ given by:

$$\mathbf{a}' = \mathbf{a} - 2\frac{\mathbf{a} \cdot \hat{\mathbf{x}}}{\hat{\mathbf{x}} \cdot \hat{\mathbf{x}}}\hat{\mathbf{x}}$$

   Rewrite $\mathbf{a}'$ to $\mathbf{a}$.

3. Choose a new data point $\mathbf{x}$ at random and go to 1.

It remains only to prove that if we keep on selecting points from the data set at random, and if the set of points is *linearly separable*, that is, if it is possible to find a line which correctly classifies the data set, then the iteration of the above process will eventually terminate with every data point correctly classified. [6] This is the *Perceptron Convergence Theorem* and it is fairly easy to understand.

Having explained the ideas, we go through the process more formally.

Definition 5.1 *A* data set *is a finite subset $\mathcal{D}$ of $\mathbb{R}^n$, together with a map*

$$c : \mathcal{D} \longrightarrow \mathcal{C}$$

into a finite set $\mathcal{C}$ of categories. *If the set $\mathcal{C}$ has only two elements, the data set is said to be a* binary data set*, and $\mathcal{C}$ is written $\{1, -1\}$ and called S^0.*

Definition 5.2 *A binary data set $\mathcal{D} \subset \mathbb{R}^n$ is said to be* linearly separable *iff there is a hyperplane in $\mathbb{R}^n$ such that all points of $\mathcal{D}$ of category $+1$ are on one side of the hyperplane, and all points of category -1 are on the other side.*

Definition 5.3 *The space of all oriented hyperplanes in $\mathbb{R}^n$ specified by the $n + 1$ weights on the arcs of a perceptron is called the* weight space.

[6] As described, it will keep going indefinitely of course, but at least it will do so in a simple way which can be checked on. Always right, it will sit smugly there for ever, so all we need to do is to monitor its torpor.

Theorem 5.1 (Perceptron Convergence) *If $\mathcal{D} \subset \mathbb{R}^n$ is a binary data set which is linearly separable, then the Perceptron Training Rule will terminate in a finite number of moves with probability 1.*

Proof. In the weight space, each augmented data point determines a hyperplane through the origin, and also a right side and a wrong side for this hyperplane, the right side consisting of the hyperplanes which correctly classify the point. The solution space of all hyperplanes which correctly classify all the data is simply the intersection of all these half spaces.

By hypothesis this is not empty, and except in a probability zero case, the intersection has strictly positive measure.

Take a ball on the origin in the weight space, and suppose, without loss of generality, that the initial state lies within this ball. Then the operation of making a change to the state of the perceptron is a folding of the ball about a hyperplane, and the inverse image by this folding of the solution space also has positive measure, and is a region of fixed positive measure removed from the set of states in the ball. Provided that all the data points operate in this way sufficiently often, the measure of the whole ball must be removed, whereupon every state in the ball is transformed to a solution state. □

It is easy to devise variants of the Perceptron training rule which are also guaranteed to give a solution from any initial state. For example, instead of the reflection, we could simply take a fixed step in the right direction, which is in the direction of $\hat{\mathbf{x}}$ if the category of $\mathbf{x}$ is $+1$ and the value of $sgn(\mathbf{a} \cdot \hat{\mathbf{x}})$ is -1, and is in the opposite direction when both signs are reversed. This will not correct the error in one move in general, but it will do so if iterated, and the idea of the proof still works: every correction takes a bite out of the space of states so that some that were wrong are now right and will never change again because they are right about all the data. And a finite number of bites of fixed size eats up any bounded set of initial states. This variant of the Perceptron training rule is known as the *delta* rule.

Note that we have the theorem in any dimension. Generalising it to more than two categories may be done in several ways, and is left as an exercise for the diligent reader. If any.

5.3 Committees

5.3.1 Committees and XOR

The XOR data set of *Fig.5.2* cannot be solved by a single unit because it is not linearly separable. It can however easily be solved by three units sharing input from points in the plane and one unit receiving its input from the first three. The diagram for the net is shown in *Fig.5.6.* It is called a committee, terminology due to Nilsson, because the last unit can be thought of as taking the votes of the three preceding units and adding them up. This will give the majority opinion since each vote is ± 1.

This committee is referred to in the literature as a 'three layer net; you might count only two layers, the square boxes being inputs not processing elements, but the traditions of the literature are against you. The fact that the literature appears to have been written by people who couldn't tell the difference between an input and a processing element may depress you somewhat, but look on the bright side. It can't really be a difficult subject.

Now we work through the points one at a time and trace the behaviour through the net. It is recommended that you do this with a pen and paper to check my arithmetic, which is very shaky.

Suppose we input the point (0,0) to the net. The first unit outputs $+1$, the second and the third -1. The fourth unit in the third layer simply sums these and outputs the sign of the result which is -1. If we input the point (1,1) then the first unit outputs -1, the second $+1$ and the third -1 again, so the last unit again outputs -1.

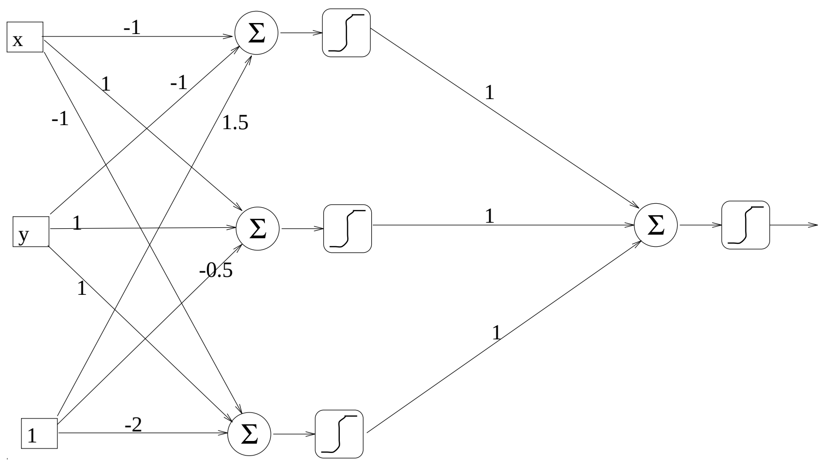

Figure 5.6: A committee of model neurons.

If we input (1,0) then the first unit outputs +1, the second unit outputs +1 and the third unit outputs −1 again. So the majority vote sum of these numbers is +1. Similarly if we input (0,1), the first two units output +1, the third outputs −1 and the last unit outputs +1. So with the interpretation +1 means TRUE and −1 means FALSE, the network implements the logical function XOR.

So a committee of three units can do what a single unit cannot. The question which should spring to the mind of even the most intellectually numb is, how did I get the numbers above?

The answer is that I drew a simple diagram of lines and read off the coefficients of the lines. The picture I drew was that of *Fig.5.7*, and I simply surrounded the positive points with some positive lines. In fact the negative line is only there as a courtesy; it seemed easier to have an odd number of voters to ensure that there can't be any ties as a general principle.

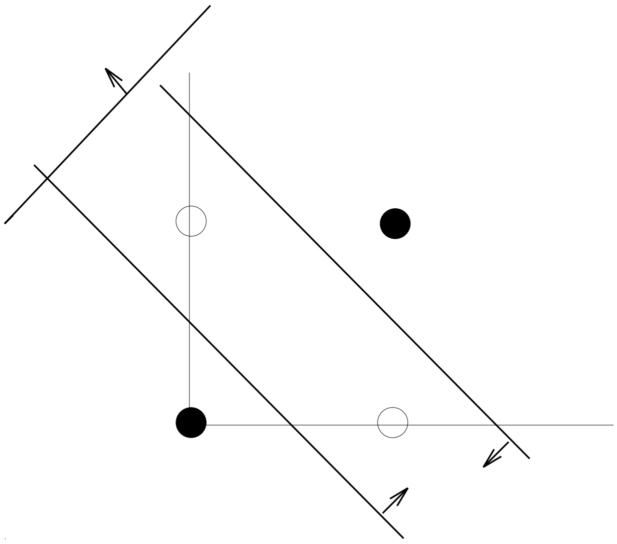

Figure 5.7: A committee solving XOR.

5.3.2 Training Committees

If the net of *Fig.5.6* is in error about the XOR data set, how do we adapt it? Let us contemplate simple heuristic arguments first, as a prelude to understanding the Back-Propagation algorithm.

First, if the net is wrong about a point, we may suppose without loss of generality that the net thinks it a -1 when it is really a $+1$. So if the weights, call them u, v, w, on the last unit are all $+1$, then two or three of the lines have the point on the wrong side. We could therefore adapt the system by taking one of the lines that is wrong about the point, and simply use the Perceptron Training rule to change that line.

As an alternative, we could change the weights u, v, w. If we had all three lines wrong, we should have to change the sign of at least one weight, which is exactly equivalent to swapping the corresponding line's orientation, while leaving it in the same place. It seems unreasonable to keep the lines fixed and merely adjust their weights, since if the lines are simply in the wrong places, they will stay in the wrong places. In fact we have simply done a fixed transform to the input and then fed the result into a single unit. We might be lucky and have done by good luck a transform which improves things, but it would be silly to bet on it. So mucking about with the weights u, v, w is not a particularly attractive option. So we may reasonably prefer to keep the voting strengths the same and at $+1$, and concentrate on moving the lines.

This raises the question of which line to move. Now a line is either right or it's wrong, but it could be argued that it can be said to be very wrong in some cases, but not so wrong as all that in others. The conservative thing to do, is to make whichever change is smaller. The rationale for this is that presumably some of the data set has been learnt by the system, even if only by chance, and that it would be silly to throw away everything, so work out which change in the state space is least and do that one. This amounts to the following strategy:

take the lines $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$ and the point $\hat{\mathbf{x}}$ such that $\mathbf{a}_i \cdot \hat{\mathbf{x}} < 0$ when the category $c(\mathbf{x}) = 1$, and find the line $\mathbf{a}_i$ such that

$$\mathbf{a}_i \cdot \hat{\mathbf{x}} < 0 \ \& \ |\mathbf{a}_i \cdot \hat{\mathbf{x}}|$$

is a minimum. Then correct this $\mathbf{a}_i$ using the Perceptron training rule. Similarly for the case where $c(\mathbf{x}) = -1$.

Experiments show that this works fairly well, and much better than such strategies as choosing the ' maximally wrong' line. A committe of three members such as *Fig.5.6* can find a solution fairly quickly in practice, for such trivial data sets as the XOR. It is easy to write a little program which draws the lines on the screen and you can watch them hop about spasmodically until they converge to a solution. There is no advantage and some disadvantage in also modifying the weights u, v, w. Higher dimensional analogues to XOR can be constructed by taking a binary string of length n and assigning the category $+1$ to it if there is an even number of 1s in the string, and a -1 if there is an odd number. This 'parity' data set has 2^n points and it is an amusing exercise to compute the smallest number of units in the second layer (committee members) required to ensure a solution.

Such committees, started off from random initial locations, can sometimes fail to converge, and a little innocent experimenting rapidly shows why. Suppose that two of the three wrong lines are 'a long way away' from the datum $\hat{\mathbf{x}}$, and indeed from all the data. Then the same line may be corrected every time a datum is presented to the net, and the two remote units might just as well not be there. The correction process ignores them. In this case we are trying to solve XOR with a single unit, which is impossible. A 'long way away' simply means that $|\mathbf{a}_i \cdot \hat{\mathbf{x}}|$ is large. This will happen for XOR if the constant term is big in absolute value.

There are several ways of fixing things up so that all units are employed usefully. One quite attractive procedure is the following: we do not simply select the closest unit for correction, we correct all of them, but we correct the closest unit most and the remoter units much less. Thus the most remote of units will eventually be pulled into the region where the data is being classified by the hyperplanes until the data is classified correctly, except where there are simply not enough units to do the job. Since we seldom know what this minimum number of units is except in rather trivial problems such as the parity problem, the prudent thing to do is to start with a lot and keep your fingers crossed.

The choice of correction rule which does this differential weighting of correction according to how remote the line is from the datum in the weight space, needs a function which starts off at some positive value and then decreases monotonically to zero. Something like e^{-x} springs immediately to mind (since distances are always positive!). It is not a particularly good choice, because it takes much longer to compute transcendental functions than polynomials or rational functions, and something like $1/(1+x)$ is easier to compute. This function has the property that remote objects get pulled in rather fast, which may be a bad idea if they are not so remote as all that, and are in fact doing a good job of classifying some other data points. One can therefore have a preference for, say, $1/(1+x)^2$. The possibilities are endless.

The end result of this purely heuristic analysis then, is that a workable correction process to the committee net can be obtained by keeping the final weights fixed at $+1$, and adapting the wrong units by taking a step in the appropriate direction (i.e. along the direction $\hat{\mathbf{x}}$ or in the opposite direction according to the sign of the category) of size some constant times a function which is decreasing to zero from 1 (or some other positive constant) and might conveniently be chosen to be e^{-x} or $1/(1+x)^2$. It is easy to explore this training rule on real data and it seems to behave itself sensibly. We shall see later that this rule is intimately related to the back-propagation procedure.

5.3.3 Capacities of Committees: generalised XOR

It is intuitively clear, and easy to prove, that *some* committee can classify any finite data set in $\mathbb{R}^n$. I do this now, so you can brood over the proof.

Proposition 5.1 *Any finite binary data set in $\mathbb{R}^n$ can be classified by some committee net.*

Proof We induct on the number of data points, N. It is clear that when $N = 1$ there is no problem, indeed it can be accomplished with one unit when $N \leq n+1$. Suppose then that all data sets having N data points in $\mathbb{R}^n$ an be classified by some committee, and consider a data set having $N+1$ points. One of these, call it p, has to be on the outside, that is must be separable from the remaining data set by a single hyperplane.[7] There is a committee net which can classify correctly the remaining N data points, by the induction hypothesis. If the same net is correct in its classification of p there is nothing to prove, so we suppose that the committee assigns the wrong category to p.

Since p is separable from the rest of the data set, we may surround p by hyperplanes which do not intersect the convex hull of the remainder of

[7]This is intuitively hard to doubt and if you wish to doubt it, do so, but be careful, you may turn into a Pure Mathematician. The price of freedom to doubt intuitively appealing assertions like this is that you get asked to prove them. Go ahead, make my day.

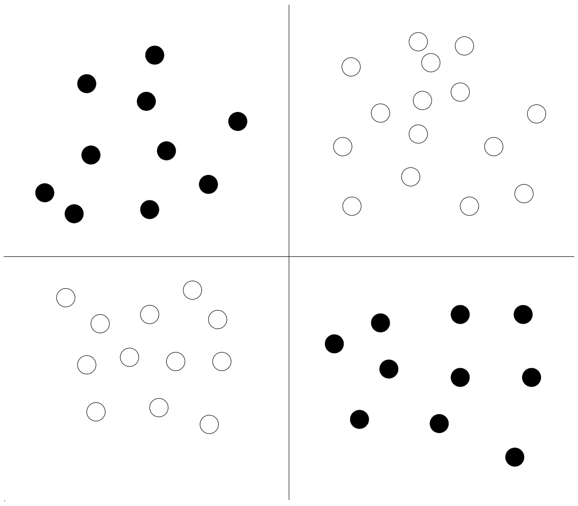

Figure 5.8: Generalised XOR.

the data set and which change the net's opinion of p, by putting them between p and the rest of the data, and we may put an equal number on the other side of p to ensure that outside this band containing p no change is made. The result is a committee which is correct about all the $N+1$ data points. The Principal of Mathematical Induction completes the proof. $\square$

Note that the size of the committee may have to be increased faster than the size of the data set. This should, after some exposure to the ideas of Rissanen, make one sceptical about the extent to which such modelling actually accomplishes anything. I shall quantify this later.

The point is brought home if one considers a more 'realistic' data set than XOR. Suppose one defines a *generalised XOR* data set in $\mathbb{R}^2$ by assuming a large number of points in the plane, all the points in the first and third quadrants have category -1 and all the points in the second and fourth quadrants have category $+1$.

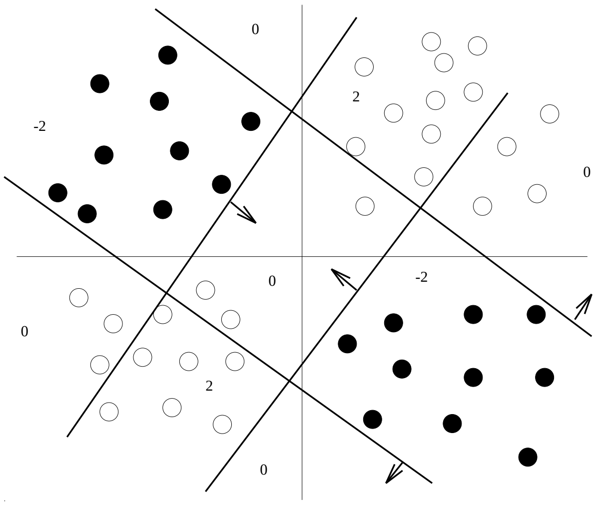

Figure 5.9: ....and a solution...

An example of such a data set is sketched in *Fig. 5.8*, where the different categories are represented by circles black and circles white. Generalisation to higher dimensions is left as an easy exercise for the diligent student. Generalisation to more than two categories is also left as an exercise for the diligent reader.

Now the smallest number of oriented lines which can correctly classify the data set looks to be, in general, on the large side. A solution to the particular distribution of points of *Fig.*5.8. is offered in the next sketch, *Fig.5.9.*

The numbers refer to the votes for and against. An additional unit or a suitably chosen threshold of 1 would make the discrimination.

Now it is easy to see that putting more data points in the four quadrants, according to the arrangement whereby points in the third and first categories are of type +1 and points in the second and fourth are of type +1, will soon start putting this model in trouble. The difficulty is quite plain: the units define lines across the whole plane, the rule for alternate quadrants has different regimes in half planes, and to continue briskly across the whole space is inevitably going to screw things up. Moreover, this is a generic problem not confined to the generalised XOR data set; if a data set consists of a collection of convex regions in $\mathbb{R}^n$, with each convex set containing points of the same category, and if the convex sets are chosen so as to make their number minimal, then if there are more than four of them, they will look like the generalised XOR set or worse.

The situation is evidently most unsatisfactory; it is easy to arive at the situation where one is trying to fit pairs of lines in so as to sneak through the regions largely occupied by the black circles, without actually enclosing any between the lines, with the intention of actually enclosing a single white circle. And the problem arises from the nature of the committee. Proposition 5.1 therefore is of limited practical value; the existence of generalised XOR and the fact that we can expect data sets which are structurally similar whenever the complexity of the pattern is high enough, tells us that three layer committees can be expected to perform badly as the number of units will be of order the number of data points. As we shall see later, the three layer MultiLayer Perceptron with a sigmoidal squashing function (and any algorithm for finding a solution) is subject to the same carping comments. The predictive power of such a system will be close to zilch. The thoughts on compression and information extraction lead us to doubt if the model implemented by such a net is worth having, a point to which I shall return later.

5.3.4 Four Layer Nets

There is a simple solution to the problem of the last subsection. We simply add another layer. The next diagram, *Fig.5.10*, gives a diagram of a net which solves the generalised XOR problem of *Fig 5.8*. Each node is supposed to do a weighted linear summation of the inputs *and* to follow this by the *sgn* function. I have left out all arcs where the weight is zero, and also the constant in the input layer, because it

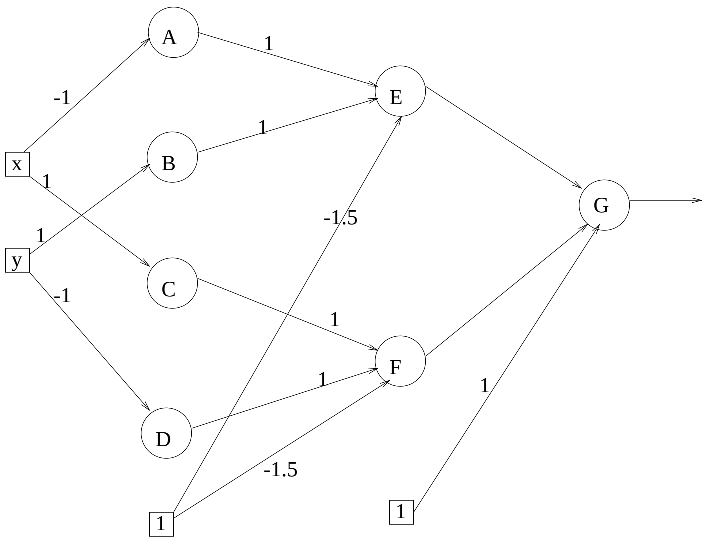

Figure 5.10: A Four-Layer net for generalised XOR

looked lonely with no arcs coming from it.

Now suppose I give you as input to the net a point in the first quadrant, that is, both x and y are positive . Then A outputs -1, B outputs $+1$, C outputs $+1$ and D outputs -1. E and F receive input sum zero and with negative threshold output -1, G has two -1 inputs which overcome the weight of $+1$, so the output from the net is -1.

If I input a point in the second quadrant, x is negative and y is positive. So both A and B output $+1$, while C and D output -1. E outputs $+1$ and F outputs -1. With that input and a positive 'threshold', G outputs $+1$, the output from the net. If the input has both x and y negative, from the third quadrant, then as in the first quadrant, E and F output -1 and the net output is also -1. Finally, in the fourth quadrant where x is positive and y is negative, C and D output $+1$ while A and B output -1, so E outputs -1 and F outputs $+1$, hence the net outputs $+1$. So for input in the first or third quadrants, the unit outputs -1, while in the second and fourth it outputs $+1$. In other words it solves the generalised XOR problem.

Inspection of the net and a little reflection shows how this is accomplished. It is useful to draw the lines in the plane implemented by the first layer units; they are indicated, somewhat displaced from their true positions in *Fig.4.11.* A and B return positive for the second quadrant, C and D for the fourth quadrant. Unit E is an *AND* gate which fires (+1) if and only if the input is in the second quadrant, unit F is and *AND* gate which fires iff the input is in the fourth quadrant. Unit G is an *OR* gate which fires if E fires or if F fires. It is easy to see that AND gates and OR gates are arranged (for any number of inputs) by judicious choice of the threshold. The effect of this is to partition the space so that we no longer have the extreme annoyance of having to put up with hyperplanes that run right across the whole of it.

Some more reflection makes it plain that it is easy to take any data set and chop it up into regions which are bounded by convex polytopes and where each region contains points of only one category. This holds in any dimension and for any number of categories.

Thus we conclude several things; first that four layer nets (three layers of processing units) suffices for any data set with the advantage over three layer (two layers of processing units) nets that the number of units required is of order the complexity of the data set, not the number of points, where the complexity bears some relationship to the geometry. This means that we do not feel any particular attraction towards nets with many more layers. The simpler models are more appealing to anyone who has read the last chapter thoughtfully.

Second, that it is tempting to fix the weights on all but the first layer, and to tackle a problem with some fixed number of convex sets ready to wrap around the regions. It is true that we do not know in advance whether we can do the job with a given number of units arranged in a particular configuration, but that is the nature of the beast. We never know, in general, whether there is a solution space for our net. There is a trade off: if we allow the weights between the second and third layers to change, we have more flexibility in the matter of the number

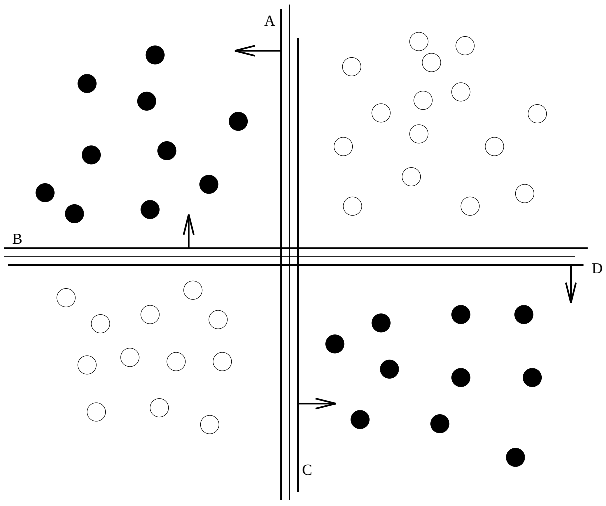

Figure 5.11: Four layer solution for generalised XOR

of units making up any polytope, and if we allow changes in the weights generally we may have more flexibility in the way the gates work, but flexibility is not always a virtue. Worms are flexible, but cockroaches can run faster. Jelly is flexible, but not the preferred medium of choice for building houses unless you are very young indeed. There is a case for having articulated rigid parts. It is found experimentally that it is possible to train an articulated neural net ten times as fast as a flexible one on problems of low complexity. The flexible ones tend to fail to converge at all in many cases. If you contemplate lines hurtling over the initial space, the next layer doing the same with the results of the output of the lines in a higher level space, and so on, you are probably inclined to find this unsurprising.

In conclusion, both three and four layer nets using a step function after a weighted summation of the input values, can classify any finite data set, but the three layer net will, in the case where the data set has non-trivial complexity, that is to say where it does not fall into bands which extend across the space, need a number of units which is of order the number of points, while the four layer net is able to do so with a number of units which are of order the *simplicial complexity* of the data set, where the simplicial complexity is defined as the smallest number of simplices which can enclose the data in such a way that inside each simplex the data points have the same category. Approaching a classification problem with the AND and OR weights all fixed can speed up convergence by an order of magnitude or more when the problem is solvable by such a net. We remind you that visual inspection of the data by a projection program can aid in deciding what class of net to use, how many convex regions are likely to be necessary and how complex each convex region may need to be. Such simple methods can reduce enormously the amount of time spent making essentially random hops around a huge weight space, a practice which has in recent times used up, one hesitates to say 'wasted', almost as much computer time as playing arcade games.

5.3.5 Building up functions

Any net implements a function from the space of inputs to the space of outputs. A single unit in $\mathbb{R}^2$, as in *Fig.5.4* is a map from $\mathbb{R}^2$ to S^0 which sends the point $\begin{pmatrix} x \\ y \end{pmatrix}$ to the value $sgn(ax + by + c)$. A three layer net such as *Fig.5.6* implements a function

$$sgn(u*sgn(a_1x+b_1y+c_1)+v*sgn(a_2x+b_2y+c_2)+w*sgn(a_3x+b_3y+c_3))$$

and it is easy to see that this is the weighted composite of functions implemented by each unit in the first layer of processing elements. This sort of thing can be carried on for as many layers as you like, although don't get too carried away just yet.

The data set constitutes a sort of function; it may be taken to be a function defined on the finite set of points and having values in S^0. And a neural net is not just *one* function from $\mathbb{R}^2$ to S^0, it is an infinite family of them, parametrised by all those weights. In other words, there is a manifold of functions, in general from $\mathbb{R}^n$ to S^0, and a data set, a function defined on a finite subset of $\mathbb{R}^n$ taking values in S^0, and the problem of training a data set is to find one of the functions which agrees with the data set function. The data set functions we consider in pattern recognition are binary valued of course, and the convention I use here is that they take values in S^0, although there are other applications where we take real valued functions. I shall not treat such cases on the grounds that you are already getting good value for money and if you have too many things to think about you might get depressed, but this section and the next two comprise a brief aside on the issues involved.

Any training procedure can be viewed as a process of starting off with some arbitrary function from the family and moving about the weight space until we get agreement, until we fit the data. In general we have no guarantee that there *is* such a function in the family. The XOR problem with one unit is an example. Here one must think of fitting step functions defined on the plane, $\mathbb{R}^2$, and taking values ± 1. It is easy to see that for three or four layer nets of the sort we have described, the complexity of the data set, however defined, is the critical consideration in whether or not the problem is solvable at all. All problems have *some* size net which can solve it; equally, all nets have some problems they cannot solve. This is not a particularly profound observation, but it raises the ante from a sort of Minskian disenchantment with the capacity of the net to other more interesting issues.

If I put the problem in the frame of function fitting from a known family of functions, then I raise immediately the question of whether the family is a good one to try to fit from. The question of what makes a sensible basis of functions for describing point sets came up in chapter two, where we were trying to describe characters, and we meditated on the merits of the Fourier basis of sines and cosines against polynomials. For a given type of data, some basis functions make more sense than others. Just choosing a basis set because it is there is not particularly intelligent; it may be that choosing it because it is easy to do the sums with it makes sense, or it may be that the kinds of functions you want to fit are likely to be well suited to the family you want to fit with. But a little thought can save a lot of computing time.[8]

The geometry of a neural net determines the function class from which you are trying to fit the data. This function class is the important thing, and the net diagram is otherwise largely uninformative, although it may make the programming task of evaluating it somewhat simpler. It determines the dimension of the space of weights through which you are proposing to step in order to find a solution. Stepping blindly may be the best you can do, but it is to be avoided if possible. It is clear that the time to find a solution goes up as the dimension of the search space, and rather worse than linearly in general. There is a trade off between finding, on the one hand that there is no solution with your net because it isn't big enough, against at least discovering, on the other hand, this uncomfortable fact fairly quickly. Bunging everything into a huge net, running it on a supercomputer and leaving it to chug away for a few weeks (or years) is the sort of cretinous stupidity one associates with......, well, not with engineers, I hope.

5.4 Smooth thresholding functions

5.4.1 Back-Propagation

The suggestion that I offered above for modifying the state of a committee by adapting all the wrong units by different amounts, more if they are less wrong, has some common sense justification, and also empirical justification: it works. Those of an idealistic cast of mind find such justifications unsatisfying, they wish to derive practice from more fundamental principles. Always nice, provided it gives the right answers, says the engineer to himself.

[8] This is a legitimate trade off, of course. The extent to which people opt for random searches through huge state spaces, rather than think about what they are doing, tells you how high a price they need to be paid in order to think. For some people, there isn't that much money.

The approach taken by Anderson, Rumelhart, Hinton and Williams was to replace the *sgn* function by a smooth approximation to it. The function tanh is favoured for reason which are not altogether compelling; it is common to use thresholding systems which output values to 0 or 1 instead of -1 or $+1$, in which case the sigmoidal approximation becomes the function:

$$sig(x) = \frac{e^x}{e^x + e^{-x}} = \frac{1 + \tanh(x)}{2}$$

It should be noted that it doesn't make the slightest difference whether you prefer 0 to -1 for representing the other state or not. If you have a predilection for 1 and 0, add one to my numbers and divide by 2.

This smooth approach has a considerable advantage from the point of view of the idealist who wants a rule for arbitrary nets. It means that the function implemented by the net is now a differentiable function, not only of the data, but also of the weights. And so he can, when the outcome is wrong, find the direction of change in each weight which will make it less wrong. This is a simple piece of partial differentiation with respect to the weights. It can be done for any geometry of net whatever, but the layered feedforward nets make it somewhat easier to do the sums.

What happens may be seen in the case of a single unit, where the function implemented is

$$n(x, y) = sig(ax + by + c)$$

which is a composite of the function $A(x, y) = ax + by + c$ with the function

$$sig(t) = \tanh(t)$$

Now partially differentiating with respect to, say, a, gives

$$\frac{\partial n}{\partial a} = x sig'(ax + by + c)$$

and similarly for the other weights.

If we choose to adopt the rule whereby we simply take a fixed step in the direction opposite the steepest gradient, the steepest descent rule, then we see that we subtract off some constant times the vector

$$sig'(ax + by + c)\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

from the weight vector $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$

which is the step rule variant of the perceptron convergence rule, multiplied by the derivative of the sigmoid evaluated at the dot product of the weight vector with the augmented data vector. A quick check of what you get if you differentiate the sigmoid function reveals that this function has a value of 1 at the origin and decreases monotonically towards zero. In other words, it simply implements the differential movement rule suggested for committees, the further away you are, the less you move. If we use tanh as our sigmoid, the derivative is $\frac{4}{(e^x+e^{-x})^2}$ which for large x is approximately $4e^{-2x}$

In the case of the three layer net, what I have called (following Nilsson) a committee net, the function implemented is

$$n(x, y) = sig(u * sig(\mathbf{a}_1 \cdot \hat{\mathbf{x}}) + v * sig(\mathbf{a}_2 \cdot \hat{\mathbf{x}}) + w * sig(\mathbf{a}_3 \cdot \hat{\mathbf{x}}))$$

where sig is the sigmoidal approximation to the sgn function and $\mathbf{a}_i, i = 1, 2, 3$ are the three weight vectors. $(\mathbf{a}_1 \cdot \hat{\mathbf{x}})$ is just $(a_1 x + b_1 y + c_1)$

If we partially differentiate this with respect to, say, a_1, we get

$$\frac{\partial n}{\partial a_1} = sig'((u*sig((\mathbf{a}_1 \cdot \hat{\mathbf{x}}))+v*sig(\mathbf{a}_2 \cdot \hat{\mathbf{x}})+w*sig(\mathbf{a}_3 \cdot \hat{\mathbf{x}}))*u*sig'(\mathbf{a}_1 \cdot \hat{\mathbf{x}})x$$

and similar results for the other parameters. Taking a step against the gradient means decreasing a_1 by some constant times this number, and corresponding amounts for the other parameters.

This is easily computed from the output backwards. If we use tanh as the sigmoid, we have the useful result that the derivative is $1 - \tanh^2$, and this means that no further calculations of transcendental functions is required than was needed for the evaluation. The generalisation to feed forward nets with more layers is straightforward. Turning the idea of gradient descent (pointwise, for each data point) into an algorithm is left as an easy exercise for the diligent student. Alternatively, this rule is the *Back-Propagation Algorithm* and explicit formulae for it may be found in the references. More to the point, a program implementing it can be found on the accompanying disk.

It is noticeable that the actual sigmoid function does not occur in any very essential way in the back-propagation algorithm. In the expression for the change in a_1 for the three layer net,

$$\frac{\partial n}{\partial a_1} = sig'((u * sig(\mathbf{a}_1 \cdot \hat{\mathbf{x}}) + v * sig(\mathbf{a}_2 \cdot \hat{\mathbf{x}}) + w * sig(\mathbf{a}_3 \cdot \hat{\mathbf{x}}))u * sig'(\mathbf{a}_1 \cdot \hat{\mathbf{x}})x$$

(where we have to subtract some constant times this) the value of $sig(a_1 x + b_1 y + c_1)$ lies between ± 1, and if the sigmoid is a steep one, that is to say if it looks like the sgn function, then the difference between sig and sgn will not be large. The derivative however does have an effect; the closer sig gets to sgn, the closer the differential committee net algorithm based on the derivative of sig gets to just moving the closest unit.

5.4.2 Mysteries of Functional Analysis

Once one has taken the step of extending finite maps having values in S^0 by smooth maps into $[-1, 1]$, the smooth maps being from some manifold of such functions, the subject starts to attract the interest of Functional Analysts, a collection of Mathematicians who, having reduced much of the theory of functional approximation to linear algebra, find themselves underemployed and reduced to manufacturing rather artificial problems, those real problems that are left being much too hard. There have been a number of theorems showing that any function from $\mathbb{R}^n$ to [-1,1] which is sufficiently well behaved (for example, continuous and having compact support[9]) may be approximated arbitrarily well by three layer neural nets. Such statements are true but of very limited practical utility. They are not offered to the engineer because they have practical utility, but because they are publishable, and they are publishable because engineers who are not familiar with a branch of mathematics are more likely to be impressed with it than they are with familiar tools. It is desirable therefore that the elements be explained in order to reduce the reader's gullibility quotient.

Whenever one talks of approximating one thing with another, one has some kind of a notion of a distance between the things, and this distance being small. More precisely, to be told that one can approximate a function arbitrarily well by polynomials is to be told that there is a sequence of polynomials, and that the further you go down the sequence, the smaller the distance between the polynomial you choose and the function you are getting closer to. And in fact the function you are approximating has to be the limit of the sequence of polynomials; the distance between polynomials in the sequence and the function you are trying to sneak up on can be made as small as anyone wants just by going far enough down the sequence. This leads us to ask how you measure the distance between functions. Suggesting, as it does, that there is one and only one 'right' way, this is actually the wrong question, but only a bit wrong.

There are many ways of measuring the distance between functions, but two are of particular importance. The first is the so called L_∞ norm, which measures the distance of f from g by taking

$$\|f - g\|_\infty = \sup\{|(f-g)(x)|\}$$

If f and g are continuous and the domain of both is compact, then this is always defined. The second is the so called L_2 norm, in which the

[9]compact support means that the function is zero except in some bounded region of $\mathbb{R}^n$.

distance of f from g is obtained by taking

$$\|f-g\|_2 = \sqrt{\int (f(x)-g(x))^2}$$

This is defined in the case where f and g are continuous and the support is compact, but in much more general cases also. If you don't know what *compact* means, the excellent book by G.F. Simmons given in the references at chapters end will tell you, along with much more material worth knowing if only so as not to be intimidated by it. Although the numbers come out as different for any two functions, the two norms are *equivalent* on the smaller space of continuous functions having compact support in the sense that if you have a sequence of functions approaching a limit function in the one sense of distance, then the same sequence will also be converging to the same limit in the other sense too. More bizarre possibilities exist, so watch out, but for the cases discussed, the different norms are not too different.

In either sense, it is often useful to be able to find out what functions can be expressed as limits of some class of functions. If, for example, we take functions on the real line which are *pdf*s, i.e. they are non-negative and have integral 1, then it might be cheering to be able to prove rigorously the statement I threw off rather casually some time back, that such a thing can be approximated as well as you wish by a mixture of gaussians. To do this, you need a careful definition of what it means to approximate a function g by some family f_p, and of course it means I can choose members of the family $f_1, f_2, \cdots f_n, \cdots$ such that the distance $\|f_n - g\|$ can be made less than any given number ε by choosing n large enough. Thus, it is easy[10] to prove that in the space $\mathcal{C}^0([0,1])$ of continuous functions from $[0,1]$ to $\mathbb{R}$ with the L_∞ norm, the family of polynomials can be used to approximate any continuous function. This is a celebrated theorem of Weierstrass.

The theorem of Weierstrass must be compared with the well known Taylor expansion of a function about a point, which for the case of infinitely differentiable functions actually gives you a polynomial and

[10] Well, fairly easy. See Simmons in the references for a perspicuous proof.

bounds on the difference between the function and the approximating polynomial. Sometimes the bounds can be estimated, which makes for a really practical theorem. You can replace one, possibly horrid, function by a polynomial *and work out the worst case discrepancy.* The Weierstrass theorem works for a larger class of functions, but doesn't tell you how bad or good a particular approximation is. (Although there are constructive proofs which *do* give the information. Functional Analysts often seem quite indifferent to these practicalities, alas.)

Moving away from the real line into $\mathbb{R}^n$ and even more awful generality, the generalisation of Weierstrass' theorem which appeals to Analysts is the *Stone-Weierstrass Theorem* which may be used to show that if any family of functions on a compact domain is an *Algebra* of continuous functions, which means that the sum of functions in the family is in, the scalar multiple of any functions in the algebra is in, and the product of any functions in the algebra is in the algebra, and if there are enough functions to separate distinct points (i.e. if there are two distinct points x and y in the domain of the algebra functions then there are functions f and g such that $f(x) \neq g(x)$) and if, finally, the algebra contains the constant functions, *then* any continuous function with the same (compact) domain can be approximated as closely as you like in the L_∞ sense. Well, a little work has to be done to bend the Stone-Weierstrass theorem to make it go, but it can be used to show that for compact subsets of $\mathbb{R}^n$, any continuous function can be approximated as closely as you like by the functions obtained using three layer neural nets with any continuous sigmoid function.

It is worth drawing some pictures here to see what is being said. If you take a square in $\mathbb{R}^2$, any single unit will implement a function that is just a smoothed version of the function that assigned $+1$ to Fred and -1 to Gladys, with a dividing line down in between them that went right across the whole plane. *Fig.5.12* tries to give an incompetent artists impression of what the function looks like, a sort of Wave Rock effect.

Now putting three of these in at the first processing layer of a three layer neural net, and then following them with one output unit, means that

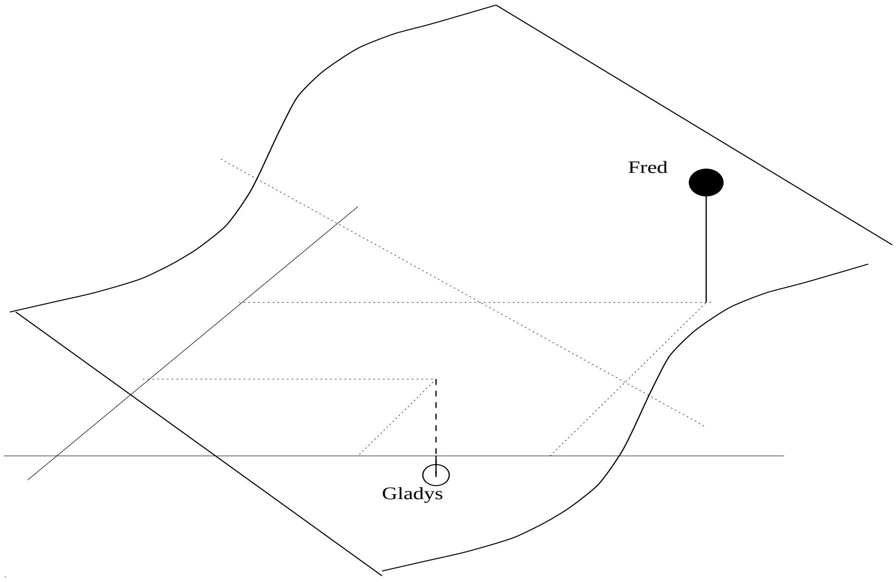

Figure 5.12: A sigmoid classifying.

we can get scaled multiples of translates of things like the Wave Rock [11]. We can even turn it upside down by multiplying by a negative number. So we have to add the result of stretching such a thing out in the x-y plane and then multiplying its height by some number and adding to the result of doing the same thing to two other such functions. It is clear that the result must have three waves reaching right across the plane. If the lines are parallel, we could use two of them to make an infinitely long ridge. Four of them could make two ridges with a sort of tower where they cross. But the waves go all across the plane. Just as we politely rejected the three layer committee net for the generalised XOR data set, on the grounds that a model where the complexity goes up with the amount of data, leaves something to be desired, so we should be rather inclined to abandon the three layer smoothed version as a suitable basis for modelling functions, unless the behaviour whereby there are waves which extend across the space were a feature of the

[11] For those readers unfortunate enough to live somewhere other than Western Australia, it should be explained that Wave Rock is a tourist attraction. It is a rock that looks quite a lot like a wave on the point of breaking

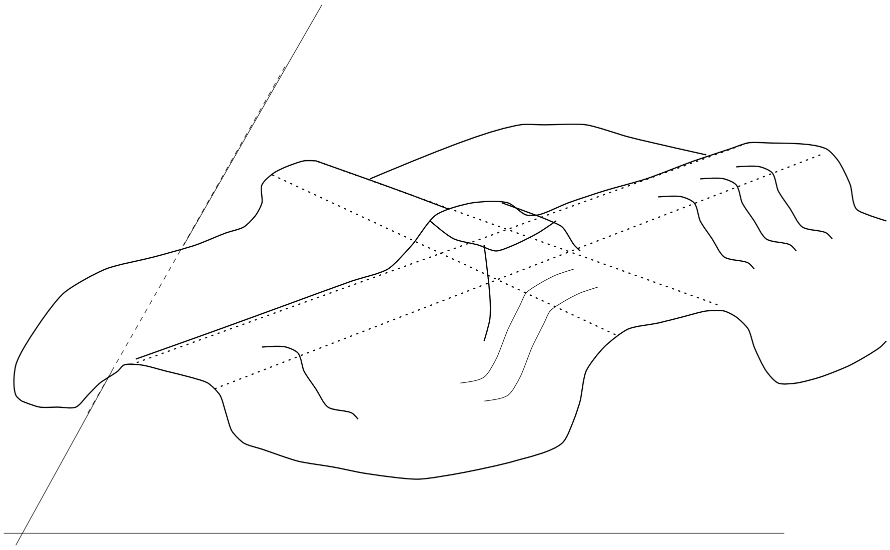

Figure 5.13: Sigmoids summing.

functions we wanted to model. The analysts claim that it can always be done is correct, but the implied claim that it is worth doing is deeply suspect. The analysts will cheerfully explain that they never said it was a good idea to actually do it, their theorems only say it can be done. The engineer will reply that they should, in that case, publish these results in some Pure Mathematical journal dedicated to useless trivia. This is an argument that can go on a long time. The moral is that although the analysts and other pure mathematicians are all God's creatures, and although they often say things that are of interest, one should read what they say carefully and very, very literally. They are not necessarily on your side.

Of course the one dimensional case is quite useful since it can be used to give a quick proof that any *pdf* can be approximated by gaussians, as you can use half a gaussian turned upside down and glued to the other half as a (sort of) sigmoid.

The theory of Fourier Analysis and the Fourier series whereby we build up a function out of sines and cosines extends the Weierstrass result in a certain sense to the case where we may use the L_2 norm, and hence to a larger class of functions. The theory can be modified to give results for three layer nets, but such results are no more practical than the Stone-Weierstrass Theorem. The question of interest is, how fast does the discrepancy disappear with the number of units used? Contemplation of *Fig.5.13* suggests that discrepancies are likely to be of the order of $1/m$, where m is the number of units This does not compare well with, say, Taylor's theorem for most analytic functions encountered in practice. Analysts are not usually interested in what functions are actually encountered in practice, and it is as well to remember this. Mathematics is a splendid servant but a thoroughly foolish master and it is essential to let it know who is the boss.

What we did with four layer nets and the hard limiter *sgn* function can be done with smooth sigmoids and function approximation, if you should feel a need to approximate functions. Since reality always gives you finite data sets to finite precision when you measure it, the need is seldom irresistible, although there are many circumstances when the procedure is useful.

5.4.3 Committees vs Back-Propagation

It is natural to wonder whether there are any advantages in using Back-propagation over a committee. The committee is experimentally found to be faster than a three layer net by about an order of magnitude and either converges fairly fast or never converges because the problem is unsolvable by the given committee. The three layer back-propagation net sometimes converges and sometimes doesn't, depending on the initial state. There is a theorem which justifies the back-propagation algorithm, sort of, but no actual proof of convergence. The state space is bigger for back-propagation, which gives more flexibility, but flexibility is a property of jellyfish and earthworms which hasn't done them a lot of good when it comes to arguing the toss with insects and vertebrates. There are more of us than of them. So one should not be unduly influenced by such simple arguments as the unbridled advantages of the larger state space.

Experiments suggest that the so called three layer net has no practical advantages over a committee of slightly larger size and an equal number of parameters, and indeed is likely to take longer to converge on average. It is fairly easy to understand the thinking behind a committee net, while back-propagation tends to numb the mind of the average user for whom calculus of more than one variable has essentially the status of black magic in the wilder parts of New Guinea. [12] Unfortunately, superstition is rife these days, and the incomprehensible is thought to have *manna* not possessed by the intelligible.

For four layer neural nets, the articulated four layer net with a fixed number of units in the second layer partitioned into groups intended to cooperate with each other in surrounding a convex region of points of some type, the partitioned groups then being ORed with each other to give a decomposition of the data as a union of convex regions, again outperforms the four layer back-propagation experimentally, being about an order of magnitude faster on sets of moderate complexity with greater relative improvement on more complex sets. Mind you, this is a competition between losers, there are better ways. It is true that sometimes back-propagation can eventually find a solution which the articulated net cannot, but at least with the articulated net you find out that there is no solution with your given geometry rather faster.

5.5 Compression: is the model worth the computation?

The purpose of using neural nets is to try to extract the maximum amount of information about the category of the data, on the hypothesis that there will be some more points along soon and we want to make the most intelligent guess at their category. If the complexity of the

[12]Or, for that matter, the wilder parts of New York.

data is sufficiently high compared with the number of points, then the simplest neural net is worse than a simple list of the points and their categories. It is true that such a procedure gives no suggestion as to what to make of the category of the next point along, on the other hand if an attempt to find stucture in the data has failed, then we might as well guess. And a data set which has shown no compression using a neural net is one where guessing merely confronts the fact that we don't have any idea of what the category of a new point is, rather than permitting us to deceive ourselves by using a neural net.

Imagine that we have a data set such as the male and female height-weight points of chapter one. It is possible to visualise an immense amount of data and to be able to believe that two gaussians, one for each category, fit the data pretty well out to three or four standard deviations. Now a four layer neural net which got every point correct would be hard to believe, and a three layer neural net utterly impossible to take seriously. All those lines which carefully snuck through the enemy camp would lead you to expect points of each category in the most peculiar places. How can we articulate our natural scepticism using Rissanen's ideas?

If we add to each datum in $\mathbb{R}^n$ its category regarded as a number either ± 1, then it takes one extra bit to do a binary classification, over the bits needed to say where the datum is in the space. Similarly, if we have gaussians doing a classification, we need to add in a bit to say which category of gaussian they are supposed to be.

In order to specify a single unit in $\mathbb{R}^n$ we need, on the face of things, $(n+1)P$ bits, where P is the precision used to specify each coefficient. It is slightly better than this in practice; multiplying the $n+1$ numbers by a positive constant will not change the output of the net or the position of the line in any way; we might therefore stipulate that the coefficients be normalised, say by making the sum of the squares of them equal to 1. This ensures that the coefficients or weights always lie between ± 1, and that if we have n of them we can calculate the last. We therefore need nP bits to specify a single unit in the second layer.

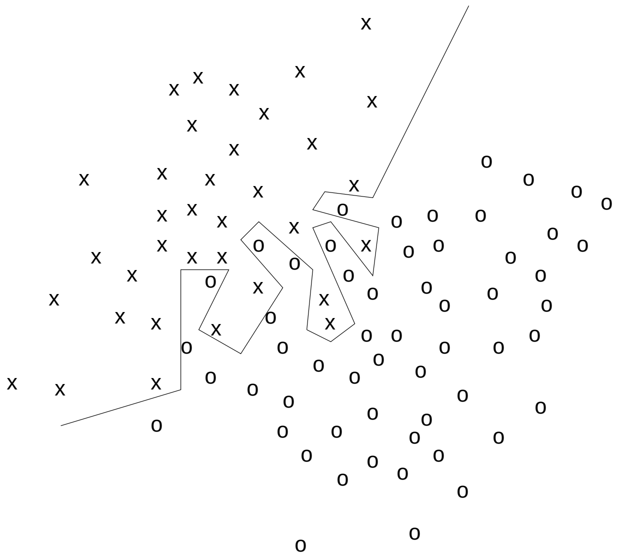

Figure 5.14: An implausibly complicated theory.

If n = 2 and P = 10, we specify the line with coefficients which are accurate to one point in a thousand. This is often excessive, since the amount of wobble of the line which can be applied and still have it implement the dichotomy may be quite large. To analyse the situation, suppose we have a binary data set in the unit square with points of category +1 at values of x and y greater than 0.6 and points of category -1 with both x and y less than 0.3. Then a specification of the position of the line needs only about three bits precision on each coefficient, if the coefficients are chosen to be between +1 and -1. To specify the data set takes some number of bits to specify the positions, say NP, and to specify the category takes a further N bits when there are N points. To use a line instead for the specification of the category, takes 6 bits, three for each (normalised) coefficient. So provided there are more than 6 data points, we have made a saving by using the line. More generally, we have that the number of points needed in dimension n in order to justify using a hyperplane is nP, where P is the precision required. If the points form two clusters which are well separated, then P goes down, which seems reasonable. There is no attempt to specify the location of the data by the model in this analysis, we only use it to store the category. If we use a committee net with k units to store the category for a binary data set in $\mathbb{R}^n$, then we have knP bits required to specify the state of the net, and again N bits for the data. If we have m categories instead of just two, we need $N \log_2(m)$ bits for the data set. In addition, there is some overhead needed to specify the network topology.

A further problem is that we do not know in advance what the precision of the coefficients of the net should be: we can put a bound on this by pointing out that it is senseless to have the precision of the coefficients much greater than that of the data, but the precision of the data is largely irrelevant in the model I propose. It is infeasible to go much further without some delicate arguments which might be thought tendentious, but it is clear that a neural net having k nodes in $\mathbb{R}^n$ must classify at least knP binary data points, for P at least of order 10, before it can be said to be extracting any structure from the data set. Papers discussing the classification of data in dimension 100 with nets having hundreds of units must therefore have about half a million data points if the application is not to be an exercise in self-deception. This criterion would exclude some well known papers in the literature. Since P is often 64 bits, and the specification of the network topology may be of order $k(k-1)$ bits, this is an optimistic analysis of the state of affairs. And there are papers in which thousands of units in dimension several hundred have been cheerfully applied to data sets having only a few hundred points in them. The naive belief that in doing this the protagonists were contributing anything to the advance of human knowledge, would appear to be sadly in error. Had they spent the computer time playing *Super Mario Brothers* they might have had more fun and not wasted the resources any more flagrantly. The figure of 10 for P agrees with rules of thumb used by the neural net community for some time, but there has been little rationale offered until recently.

It is important to understand then that when you read a paper describing a neural net application in which there are fewer than knP binary data classified by a net, the writer has not used his net to extract information from the data. On the contrary, he has put information in.

The extra information is not about the data, it is about the deficiencies of his education.

It is very hard to believe that a model of a process which has failed to extract any information from the data given, will have any real capacity to predict the results of more data measurements. In the case of neural nets which, like Feed-forward Back-Propagation nets, start with some random initialisation, then there is not, in general, any unique solution. Moreover, the less information is extracted from the data in finding a solution, the larger the range of solutions there will be. Now two different solutions will disagree with each other in their predictions, by definition, and the larger the space of different possible solutions, the greater will be the extent to which solutions can disagree with each other about the result of classifying a new datum. We cannot therefore feel any confidence about any one prediction which such a net provides, because we know that there are other solutions, just as compatible with the data, which yield other, contradictory, predictions. The extreme case occurs when no compression takes place, and so no information has been extracted from the data: for a binary data set it is possible to persuade oneself that the average area of disagreement between models which differ only in their initial state can be about half the measure of the states which data can occupy. In this case, to use any one of these models is a somewhat expensive equivalent of tossing a coin. Instead of initialising a coin with an unknown initial velocity and momentum, one has initialised a back-propagation network with random values, and the resulting prediction for a new datum depends on just which initialisation was chosen. Coins are quicker, cheaper and no less likely to give the right answer. Anyone who has any kind of model fitted to data where there are less data than parameters to be fitted, is deceiving himself if he imagines that his model has any useful predictive power, or indeed any value other than as an indicator of incompetence. All he is doing is the equivalent of simulating a coin toss, with the drawback that it takes much longer and wastes a deal of computer time.

5.6 Other types of (Classical) net

5.6.1 General Issues

It is worth contemplating the nature of the neural net problem again. As mentioned in Chapter one, supervised learning means function fitting. We have some quantity of data which are represented as points in $\mathbb{R}^n$ for some n, and we have values assigned to each of these points which may be real values or may be the categories to which the points belong. A Neural Net has to be prepared to say what the value or category is for some new point it hasn't seen before. In other words, it has to assign a value to each possible point of the space which is a potential datum. Which means it is a map or function defined on the space $\mathbb{R}^n$, or a piece of it, which agrees with at least some of the data values, more or less.

Unsupervised learning, by contrast, means finding clusters in the data, deciding when two things belong in the same category without being told what the categories are. The space may be discrete, as when we are looking at 11 by 9 grids of binary pixels, the case where we give a list of different characters without saying which each is supposed to be and ask the system to pick out the different letters without knowing what they are. Or it may be a continuum. Either way we can represent the data as points in $\mathbb{R}^n$ for some n. And inevitably we run into problems of what 'close' means. Clustering is a complicated business which can be accomplished in lots of ways, with nuances I have not discussed. But the two problems although related are distinguishable, and the basic framework is worth bearing in mind in what follows. Unsupervised learning is also referred to as the behaviour of a *self organising system* in the literature. I suppose there is an image, invoked by this terminology, of a chemical broth evolving through amino acids to proteins to a tribe of jelly fish. Since we are interested in winding up with a program that can respond usefully to a new datum such as an image, and are not proposing to simply watch on our computer screen a sort of Conway's Life game, evolving Truth and Beauty, as a substitute for

Saturday afternoon football on television, our interest in self organising systems is confined to finding clusters in the data. What is meant by a 'cluster' is left deliberately vague for the moment, but classifying human beings into males and females can be done reasonably well in the height-weight diagram by noting that two bivariate gaussians seem to fit the data in a natural way. And it isn't too unreasonable to suppose that if we measure a rather larger number of parameters by looking at and listening to them, we might find two largely discriminable clusters in the larger representation space. So although we wouldn't have the names of the categories, we would have a system set up for doing a classification or recognition job by concluding that there are some natural categorisations of the data, the clusters.

There are certainly subtleties to consider at a later stage, but it is as well to focus on these two matters first when considering what a neural net of some type accomplishes. Thus life is made simpler when one works out that the Kohonen Self-Organising Map (SOM) is finding a cluster with some sort of manifold stucture, and the ART of Grossberg is also a cluster finding system. The nets which do associative recall have the problem of mapping new points to one of the older points or some output associated with the older points on which the system has been trained.

Another distinction which may be of practical importance is the 'on-line/off-line' classification. Is the data given all at once, or is it coming in serially, one datum at a time, or is it coming in in lumps of many data at once, but not all? Most statistical methodologies presuppose that all the data is there to be operated on in one pass, in order to do either the function fitting or the clustering, while most neural net methodologies assume the data is coming in sequentially. The neural netters therefore tend to use a dynamical process for generating a solution, something like a Kalman Filter, which starts with a possibly random initial estimate and then updates it as the data comes in down the line. Bayesian methods lend themselves to this kind of approach, and other methods can be adapted to it. This leads us to study dynamical systems which find clusters, or which fit functions to data.

In talking about neural nets in this way, as things which find clusters or fit functions from a family to data, I may be making you nervous. This level of abstraction makes many people extremely unhappy; they feel insecure without concrete entities they know and love. Such people concentrate on details and fail to see the wood for the trees. Their explanations of what they are up to are confusing because it is hard to see what problem they are trying to solve or what kinds of methods they are using. Abstraction for its own sweet sake may be a sterile game for losers, but being able to figure out what *kind* of thing you are doing can help you do it better, and also allow you to see that two methods that appear to be different are really the same with the names changed.

So as to establish that I am not the only one who feels this way, I have deferred to my friend and colleague Dr. Christopher deSilva who has written the second part of the survey of other people's neural nets. We thought it wise that he should do this part because he is much more restrained and polite [13] than I am, thus keeping down the costs of actions for libel.

5.6.2 The Kohonen Net

Inspired by some results obtained in Neurophysiological studies, where it has been shown that there is a mapping between the world and adjacent neurons in the human cerebral cortex, Kohonen in the 1980's showed how a set of neurons could plausibly be constructed so as to adapt to extract the shape of a data set.

Suppose that we have a set of points in the plane that form a curve, lying, for example roughly equally spaced along the curve $y = x^2$ for definiteness. Suppose we have a line of model neurons arranged initially at random in the same space. Let the line consist of 'neurons' $\{n_1, n_2, \cdots n_k\}$, where each neuron is simply a point in the space, but where the consecutive points are imagined as joined by line segments.

[13] And, let's face it, *civilised.*

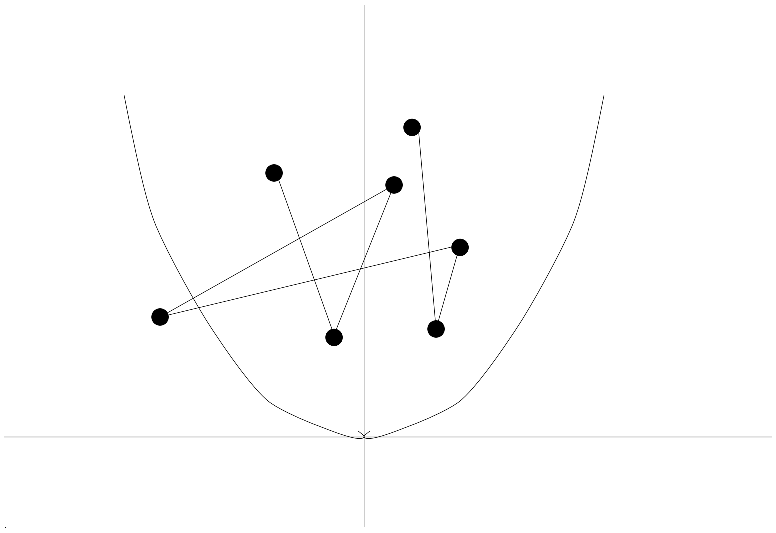

Figure 5.15: Initial (random) state of Kohonen network.

Present a point in the data set, x_j to the network. The Kohonen rule is to move the closest point of the set of neurons towards the calling datum, but also to move, by a smaller amount, the adjacent neurons. Adjacent not in the sense of being close in the plane, but in the sense of being neighbours in the ordering of the neurons, not at all the same thing in general. We can imagine the ordered line of neurons as being, initially, badly tangled by its random embedding in the plane.

If this process is iterated, with different choices of the data points selected to do the calling, the tangle of neurons gradually becomes straightened out and the line is attracted towards the data points and becomes an approximation to it. The two senses of 'close', close in $\mathbb{R}^2$ and close in the topology of the space of neurons, become similar. Mathematically speaking, the cleanest way to look at this is to think of the tangled grid of neurons as being the image of a continuous map, f_0 from the standard $k-$cube into $\mathbb{R}^n$, or in practice from some finite approximation to such a $k-$cube consisting of a k-dimensional grid. Then a single point of the data set in $\mathbb{R}^n$ is presented to the system, and the map is modified by being made to move the points of the $k-$cube closer to the calling point, in such a way as to preserve the continuity of the map. Thus we have the whole data set acting on the continuous maps from the $k-$cube into $\mathbb{R}^n$ so as to make them fit the data set. The sequence of maps should converge to some satisfactory map eventually, if the algorithm is right.

Kohonen wrote of the 'order' being preserved, although it could also be reversed by the process, if one takes it that the data set is also ordered. It is simpler to note that the mapping from the line of neurons to the line of data points is locally 1-1 or is an *immersion* in the jargon of the topologists. At least it would be if we had an infinite number of neurons which really lay along a line.

It is not difficult to generalise to more complicated objects than curves being fitted with a linear grid. If one has a set of data points in $\mathbb{R}^n$, it may be that they form a random cluster in the space, or it may be that they form a lower dimensional set. For example, they could be a swarm of flies all of which are sitting on a large inflated balloon; in this case the dimension of the surface of the balloon is two, less than the original three in which the flies are free to move. A fly which chooses to move only on the surface of the balloon is giving up a whole dimension, just like Obi Wan Kenobi in the *Star Wars* epic.

The set of points on the surface of a spherical balloon constitute a *manifold* and the flies form a finite subset of this manifold. One can imagine linear submanifolds of $\mathbb{R}^n$ or curved ones, as in the case of the balloon. If there are enough flies, it is easy for the eye to discern the lower dimensional structure, or at least the eyes, since some parallax is necessary before it can be seen. A manifold of dimension k is a generalisation of the idea of a surface such as a balloon which is technically described as a 2-manifold. A k-manifold is said to be *parametrised* when there is some collection of k-dimensional disks which are mapped to it so as to cover it without folding. Conceptually like sticking postage stamps in order to cover a spherical balloon, such parametrisations are usually given by explicit maps, an example being the map that sends the interval $(-\pi, \pi)$ onto the unit circle by sending t to $(\cos(t), \sin(t))$. This covers every point of the circle except the point (-1,0), so we send another copy of $(-\pi, \pi)$ to the circle by the map that sends t to $(-\cos(t), \sin(t))$, which has a lot of overlap with the first map. Each map separately is $1-1$ and infinitely differentiable, we have covered or *parametrised* the circle with a pair of one dimensional disks (open intervals). A Sphere could likewise be parametrised with a couple of open 2-disks, and it is an easy exercise to find a couple of explicit maps from the interior of D^2 to $\mathbb{R}^3$ which give a parametrisation of the 2-sphere.

Manifolds occur naturally because they arise whenever the n variables which specify the results of making some measurements are not independent but are related in some way. In the case of a sphere in $\mathbb{R}^3$, we have three variables x, y, z and the relation $x^2 + y^2 + z^2 = 1$, and the relationship is given by a smooth function. To see how this might generalise to a physical system, consider a 'speech space' consisting of points each of which is a 32 msec segment of an utterance by a human being. If we were to take a FFT of the waveform, we might get a point in $\mathbb{R}^{512}$ giving the energy in 512 frequency intervals. Now there are constraints involved in a human being making sounds, constraints given by the shape of the vocal tract and the ways in which it can be moved. It is plain that not all possible sounds can emerge from a human being's vocal tract, and that there may be limits upon where the points of speech can fall, and in which directions, from a given segment of speech, it can move. So it is conceivable that the speech space constitutes a manifold, or an approximate manifold, of dimension much less than 512. As is indeed the case. The pictures in some of the papers cited in the bibliography show two dimensional grids trying to fit to a cluster of speech points.

More generally, then, if we have a set of data points in $\mathbb{R}^n$, lying on a manifold of dimension m, and if we have a grid of model neurons arranged to lie in a cube of dimension k, the Kohonen algorithm may be easily modified to attract the grid to the data and to do as good a job of fitting the data as possible. If $k < m$ then it will resemble the case of a line of neurons trying to fit itself onto a plane, and a deal of folding will be necessary and the final fit less than adequate; if $k > m$ then it will resemble the case of a square being attracted to a curve, and a good deal of buckling will be necessary, but if $k = m$ then a relatively good fit may be obtained. The general property that the k-cube fits itself without local folding to a k-manifold may be shown to hold, although the term 'order preserving' is inappropriate here. Experiments show that manifold structure can be extracted auomatically by this means. See the references for *Dimension of the Speech Space* as a case in point, and *Kohonen's algorithm for the numerical parametrisation of manifolds* for a discussion of the essential properties of the Kohonen algorithm. It is accomplishing, for a finite point set which comes from a manifold, a sort of parametrisation of part of the manifold, given by a map of the k-disk (or k-cube, or grid) into the containing space.

We can see this as a sort of structured clustering: if there is a manifold structure instead of a noisy cluster structure this system will find it. But what if both occur? As soon as one starts to think about the possibilities for what sorts of shaped subsets of $\mathbb{R}^2$ or $\mathbb{R}^3$ we can have made up out of finite point sets, one starts to boggle a bit at the prospect of using any one system. And $\mathbb{R}^n$ for higher n leaves the imagination behind fast.

The general device of finding a point (e.g. a zero of a function) by making the point an attractor of a dynamical system goes back to Newton and the Newton-Raphson method. In the present case, the attractor is a set of points rather than just one, and some aspects of the topological structure is being extracted. It is a method which has some charm, and is suceptible of a range of applications; one of the few cases where an idea from the study of Artificial Neural Nets has applications outside its original domain. Chaos theorists are often presented with finite point sets in $\mathbb{R}^n$ which purport to be from a strange attractor, and trying to extract its dimension (not in general an integer, since attractors don't have to be manifolds and usually aren't) so the study of point sets and their shapes is flavour of the month in Dynamical Systems theory as well as for us humble Pattern Recognisers.[14]

The applications to Pattern Clasification are more doubtful however,

[14]Some of us are humbler than others, of course. And some of us have a self-esteem that would survive impact by a medium size asteroid.

and the differences between the piecewise affine neural nets and the Kohonen nets are large. But the basic idea has novelty and is not immediately obvious. There are lots of ways of extracting linear or affine strucure in data, but when it is non-linearly arranged in the space, there are few methods available and the Kohonen process is one of them.

The Kohonen process has been used to fit a two dimensional grid to the space of speech sounds, and a trajectory in, perhaps, $\mathbb{R}^{16}$ is thus reduced, by choosing nearest grid points, to a two dimensional trajectory. The reduction may easily seem excessive, and doesn't do a lot for automatic speech recognition, but there is clearly a good idea in there somewhere.

5.6.3 Probabilistic Neural Nets

Donald Specht published in the 1990 issue of *Neural Networks* mentioned in the bibliography a paper called *Probabilistic Neural Networks*. This followed earlier work which was not widely known to the Neural Net community which dates from the later sixties. The earlier work talked of polynomial discriminant functions and didn't mention neural nets which indeed were barely off the ground in those days.

One of the problems associated with MLPs, which is complained about from time to time by the more thoughtful users, is that whenever there is much structure in the data, the wildly hopping hyperplanes do not do a good job of collaborating to find it. A well known test case is the so called *double spiral* data set, shown in Fig.5.16.

The prospect of training a four layer neural net to learn this arrangement of data is depressing to anyone who has ever tried, and also to the even smaller group who have thought about what it entails.

The difficulty of course lies in the restrictions of thought which Neural Netters have imposed upon themselves. Some of these restrictions arise from a training in mathematics which has emphasised the formal at the

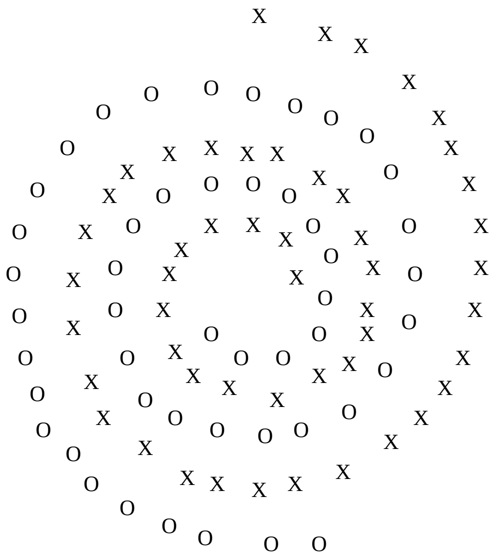

Figure 5.16: The Double Spiral Data Set.

expense of the intuitive; some from an inability to abstract the essentials of a problem. What a Multi-layer Percepron does is to chop up the space into chunks, each chunk of which contains points from the data set of the same class. There is no compelling reason why the chunks should have boundaries which are parts of hyperplanes unless there is compelling reason to think that real neurons do in fact implement such sets, and there isn't. So we can choose any set which is convenient. Specht elects, in effect, to put spheres around the data points, and to label the spheres with the class of the data. He then takes the radius of the spheres as a parameter to be played with, and pumps it up or down until it interferes with spheres of a different class. He has the same radius for every sphere; he can interpret the sphere as one standard deviation of a spherical gaussian distribution and claim that he has a sort of gaussian mixture model for the data, constrained so as to have a covariance matrix which is some constant times the identity matrix. This allows him to not only classify the data, but also to have a probabilistic assessment of the category of another point. All the algorithm actually does is to take each data point as it comes in and centre a sphere on it. Then there is some messing about to decide what the radii of the spheres ought to be. By interpreting the sphere as one standard deviation of a gaussian distribution, it is possible to compute likelihoods for subsequent data. One usually avoids thinking of the geometry by describing the process by means of a covariance matrix which is some constant times the identity matrix, but only a small amount of thought shows that this defines a sphere, with the constant specifying the radius. It is easy to visualise the algorithm applied to the double spiral problem, and this allows one to see its merits and drawbacks rather clearly.

Parzen had earlier (1962) suggested essentially the same thing with a variety of different functions replacing the spherical gaussian.

The function fitting aspects of this procedure are worth developing; if instead of a finite number of categories each data point is assigned a real number, we can take a gaussian function centred on some point and multiply it by a number just sufficient to make the value of the scaled gaussian equal to the value assigned to the point. If we do this for every point in the data set and also pump up the standard deviation of the gaussians, we may get a reasonable approximation to a smooth curve or function through the data. Fig.5.17 shows this done for the case where the data is in $\mathbb{R}$; the original values are given by spikes of the right height. We are, of course, under no compulsion to use gaussians, and function fitting by a generalised version of this is well known under the title of *radial basis functions*.

Putting a gaussian at each point of the data is rather wasteful, and this leads to trying to adapt the centres of the spheres so as to be somewhere in the middle of clusters of data and to have rather fewer of them. This is called using Adaptive Probabilistic Neural Nets, which in tune with modern methods of causing confusion are abbreviated to APNN. I shall say something about how to accomplish this later.

It may be asked what Adaptive Probabilistic Neural Nets have to offer over gaussian mixture modelling and the answer appears to be largely as given in the preamble to this chapter. There are packages one can

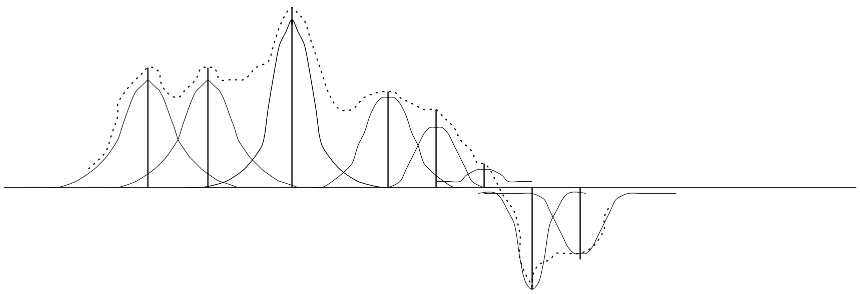

Figure 5.17: Radial Basis Functions, a.k.a. PNNs fitting a function

use, or algorithms one can copy without the least understanding of what is actually going on. Function fitting or modelling probability density functions sounds, perhaps, less glamorous than supervised and unsupervised learning by Neural Nets. This is silly and indicates the naivety of much work in the area. Perhaps the reason that many writers in the area are obscure is because it conceals the poverty of the ideas. Where there are implementation issues, such as making PNN out of hardware or the (remote) possibility that they model real wetware neurons, these strictures of a professional grouch may be lifted somewhat, but one cannot help wishing that Mathematicians had done a better job of exposition of the ideas of their subject when teaching Engineers.

Probabilistic Neural Nets are not attractive from the point of view of their Rissanen complexity, since you get to send the whole data set plus at least one number specifying the radii of the gaussians. Adaptive PNNs are more attractive, and there are justifications of them which I shall consider later in the section on Quadratic Neural Nets, which contain APNNs as a special case. The network diagrams for PNN's do not convey much information except that it is possible to implement such things via parallel algorithms, which may or may not be an advantage. If the data looks like the double spiral data, one might wish that some acknowledgement of relationships between the bits were possible. But we shall return to such issues when we come to Syntactic Pattern Recognition.

5.6.4 Hopfield Networks

Note.[15]

Introduction

Hopfield networks were introduced in the early 1980's by John Hopfield, a well-known physicist. They originated as neurophysiologically-motivated models for **content-addressable memories** (CAM).

Content-addressable memories differ from conventional computer memories in that information is retrieved from a CAM by specifying either a part of the desired information or some properties of it, instead of specifying the location in the memory where the information is stored. This is reminiscent of the way in which human memory works; for example, we can recall "a long word beginning with the letter 'A' meaning *capable of using both hands with equal facility*".

Network Characteristics

The Hopfield network works with input patterns which are vectors of real numbers. In his original paper [1], the input patterns consisted of binary vectors: each component was either 0 or 1. We will use $+1$ and -1 instead; this simplifies the algebra. In a subsequent paper [2],

[15] The following sections have been written by Dr. Chris deSilva

Hopfield extended this to the case where the input patterns are vectors of numbers from some closed interval $[a, b]$.

In all cases, the units of a Hopfield network have states, which are described by numbers belonging to the set of possible pattern values. In the binary case the states are either 0 or 1, or $+1$ and -1. In the continuous case, the states are members of $[a, b]$. Patterns are input to the network by setting the states of the units to the appropriate values, according to some mapping from the components of the input vector to the units.

The Hopfield network is not trained in the way a Backpropagation network is. Instead, it resembles the Probabilistic Neural Network of Specht in that a set of exemplar patterns are chosen and used to initialize the weights of the network. Once this is done, any pattern can be presented to the network, which will respond by displaying the exemplar pattern that is in some sense similar to the input pattern. The output pattern can be read off from the network by reading the states of the units in the order determined by the mapping of the components of the input vector to the units.

The topology of Hopfield network differs from those of the two networks mentioned in the previous paragraph. There are no distinct layers; every unit is connected to every other unit. In addition, the connections are *bidirectional* (information flows in both directions along them), and *symmetric:* there is a weight assigned to each connection which is applied to data moving in either direction.

The figure shows a Hopfield network with six units.

Network Operation

As indicated above, the initialization of the Hopfield network stores a number of patterns by encoding them in the network weights. An input pattern is fed to the network by setting the states of the units to the levels determined by that pattern. The units are then allowed to

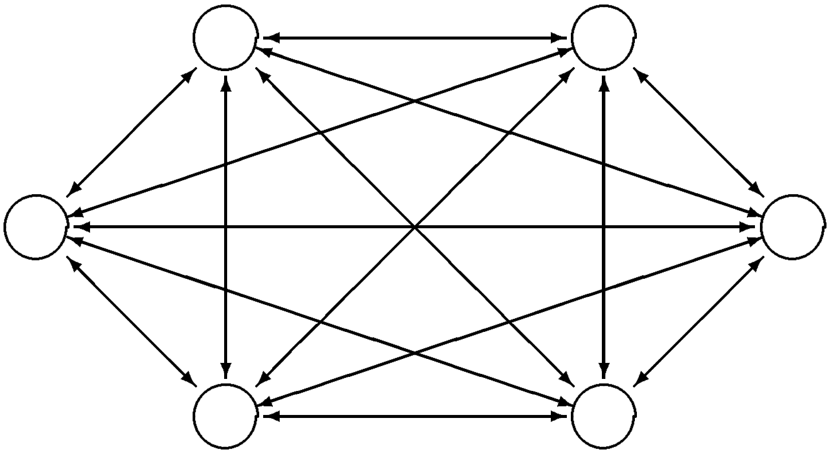

Figure 5.18: A simple Hopfield network

interact as described below until the interaction reaches some steady state. This steady state will be one of the exemplar patterns, expressed in terms of the states of the units. (It is possible that the network will not attain a steady state, but will cycle through a sequence of states. We will not consider this case in the discussion to follow.)

The interaction between units is governed by the weights on the network connections. The weights must be set to values that will ensure that the eventual steady states are the ones representing the exemplar patterns. How these values are determined will be described below.

The interaction between the units can be thought of as a sort of competition between the units in which some units excite others, making them more active (that is, increasing the numbers describing their states), and some units inhibit others, making them less active (decreasing the numbers describing their states). The choice of weights ensures that these changes do not continue indefinitely.

An alternative way of considering the network is provided by a very simple model. Consider a billiard table whose surface is not flat, but slopes towards the pockets. The surface of the table can be taken to represent the possible states of a network with two units, and the pockets the positions of exemplar patterns. Supplying the network with an input pattern corresponds to putting a ball down on the table. The interaction between units until a steady state is reached is like the ball rolling down the sloping surface of the table and falling into a pocket.

The Network Equations

In general, let the network have N units, numbered from 1 to N. The weight assigned to the connection from the ith unit to the jth unit is w_{ij} for $1 \leq i, j \leq N$. Since no unit is connected to itself, $w_{ii} = 0$ for all $i = 1, \ldots, N$. Since the connection from the ith unit to the jth unit has the same weight as the connection from the jth unit to the ith unit, $w_{ij} = w_{ji}$.

Each unit has a threshold associated with it. The threshold determines the change in the state of the unit in response to inputs from the other units. We will denote the threshold of the ith unit by θ_i.

Suppose we have M exemplar patterns, $\mathbf{p}_1, \ldots, \mathbf{p}_M$, each of which has N components, so $\mathbf{p}_i = (p_i^1, \ldots, p_i^N)$ for $i = 1, \ldots, M$. We compute the weights according to the formula $w_{ij} = \sum_{k=1}^{M} p_k^i p_k^j$ for $1 \leq i, j \leq N, i \neq j$.

Let the state of the ith unit at time t be $\mu_i(t), i = 1, \ldots, N$. Suppose we have an input pattern $(x^1, \ldots, x^N)$ that is to be classified. We impose the pattern on the network by setting $\mu_i(0) = x^i, i = 1, \ldots, N$.

To run the network, we compute the states of the units at successive instants using the formula

$$\mu_i(t+1) = f\left(\sum_{j=1, j\neq i}^{N} w_{ij}\mu_j(t) - \theta_i\right)$$

for $i = 1, \ldots, N$ and $t = 1, 2, \ldots$

In the case where the units have binary outputs, f is the step funtion: $f(x) = -1$ if $x < 0$ and $f(x) = 1$ if $x > 0$. (If $\sum_{j=1, j \neq i}^{N} w_{ij} \mu_j(t) - \theta_i = 0$ the state of the unit is not changed.) In the continuous case, f is a sigmoid function with range $[a, b]$.

Theory of the Network

The theory of the network is an extension and formalization of the billiard table model described above. Instead of the surface of the billiard table, we consider an energy surface over the state space of the network.

The state of a network with N units at time t can be described by a vector $(\mu_1(t), \ldots, \mu_N(t))$, where $\mu_i(t)$ is the output of the ith unit at time t. We define the **energy** at time t to be

$$E(t) = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} \mu_i(t) \mu_j(t) + \sum_i \mu_i(t) \theta_i$$

where θ_i is the threshold of the ith unit.

The energy of the network is analogous to the potential energy of a physical system. In the billiard table model, it corresponds to the difference between the height of the surface of the table at a point and the height of the nearest pocket. The choice of the energy function makes the Hopfield network essentially the same as a Ising model for spin glasses, which makes it of great interest to physicists.

The operation of the Hopfield network may be summed up as energy minimization. The choice of the network weights ensures that minima of the energy function occur at (or near) points representing exemplar patterns. The formula used to update the states ensures that the energy of each state is no greater than the energy of the preceeding state. The network "rolls down the energy surface to the nearest exemplar pattern".

There are two methods for updating the states of the units. In the *sequential* method, the states of the units are changed one at a time, in random order. In the *simultaneous* method, the states of the units are all changed at the same time, on the basis of their states after the previous update.

It is easy to show that the sequential updating method decreases the energy at each step. Suppose that at time t, the state of the kth unit is changed. If the change in the state of the ith unit is $\Delta\mu_i(t+1) = \mu_i(t+1) - \mu_i(t)$, then we have $\Delta\mu_i(t+1) = 0$ for $i \neq k$, and $\Delta\mu_k(t+1) \neq 0$.

The change in the energy caused by the change in the state of the kth unit is

$$\Delta E = -\frac{1}{2}\sum_i \sum_j w_{ij}(\mu_i(t+1)\mu_j(t+1) - \mu_i(t)\mu_j(t)) + \sum_i \theta_i(\mu_i(t+1) - \mu_i(t)).$$

Adding and subtracting $\mu_i(t+1)\mu_j(t)$, and simplifying, we get

$$\Delta E = -\frac{1}{2}\sum_i \sum_j w_{ij}(\mu_i(t+1)\Delta\mu_j(t+1) + \Delta\mu_i(t+1)\mu_j(t)) + \sum_i \theta_i \Delta\mu_i(t+1),$$

which can be reduced to

$$\Delta E = -\Delta\mu_k \left(\sum_i w_{ik}\mu_i - \theta_k\right),$$

using the facts that $\Delta\mu_i(t+1) = 0$ for $i \neq k$ and $w_{ij} = w_{ji}$.

The term in brackets is the same as the argument of the threshold function of the kth unit, so if its value is positive, the value of $\Delta\mu_k(t+1)$ is also positive, so ΔE is negative. If the value of the argument of the threshold function is negative, so is $\Delta\mu_k(t+1)$, and again ΔE is negative. Hence each change of state brought about by the sequential updating method reduces the value of the energy function.

The simultaneous updating method also leads to a reduction in the energy at each stage, but this is more difficult to show.

Since the operation of the network ensures that its energy is reduced at each update, it follows that the network will tend towards states that are local minima of the energy function. The positions of these minima are determined by the weights w_{ij}. The formula used to determine the weights from the exemplar patterns ensures that local minima of the energy function will occur at positions corresponding to the exemplar patterns, provided the number of exemplars is small in comparison to the number of units. It is usually stated that the number of patterns that can be effectively stored in the network is of the order of 15 per cent of the number of units. This makes the Hopfield network relatively inefficient as a memory device.

Applications

The principal application of the Hopfield network has been by physicists to the theory of spin glasses. There have been uses of the network for associative or content-addressable memory, in spite of its inefficiency. The architecture has also been the inspiration for networks which minimise functions other than the energy function introduced by Hopfield.

An application of Hopfield networks to the problem of recognizing handwritten digits is described in [3], as part of a comparison of the performance of a number of network architectures on this problem.

5.6.5 The Boltzmann Machine

Introduction

The energy function of the Hopfield network is determined by the exemplar patterns. The network weights are fixed by the patterns and there is no means of having the network learn the weights through exposure to them. The Boltzmann machine extends the capabilities of the Hopfield network by introducing an algorithm for the adaptive determination of the weights.

The Hopfield network finds local minima of the energy function. In many cases, this is sufficient, but in many cases, too, it is desirable to find the state which is the global minimum of the energy function. The Boltzmann machine combines the Hopfield network architecture with the process called **simulated annealing** in an effort to find a global minimum of the energy function.

Simulated Annealing

Annealing is a process used in glass making and metallurgy. It is the process of solidifying glass or metal from the molten state by cooling it slowly. Slow cooling results in a far more orderly arrangement of atoms and molecules than rapid cooling, with the result that the resulting solid is far stronger.

In minimization problems, a basic obstacle to the determination of the global minimum is the presence of local minima. Iterative methods for seeking a minimum, such as hill climbing, can get trapped in a neighbourhood of a local minimum, because escaping from such a neighbourhood would involve a temporary increase in the value of the function that is to be minimized.

The solution to this problem proposed by simulated annealing is to suspend the minimization process occasionally, in the hope that this will allow the process to escape from a local minimum to some other minimum, and possibly to the global minimum. The process is controlled by a parameter which is varied during the course of the process is such a way as to ensure that escape may occur frequently in the early stages of the process, and less and less frequently as the process continues. This parameter is the analogue of the temperature in physical annealing, where the temperature controls the mobility of atoms and molecues, making them less mobile as the substance cools.

Simulated annealing is an iterative minimization process in which the system moves from state to state, subject to the temperature parameter. At each stage, the system can move either to a state for which the energy is higher than its present state, or to a state of lower energy. The decision as to which state is made randomly, the probabilities of each being determined by the temperature parameter. Early in the process the temperature parameter is high, as is the probability of moving to a state of higher energy. As the process continues, the temperature parameter is reduced, and with it the probability of moving to a state of higher energy. This procedure is applicable to a number of different minimization procedures, and has been found to be effective in finding the global minimum or minima close to the global one.

Network Characteristics

There are a number of different descriptions of the Boltzmann machine in the literature. The following is based on the paper of Ackley *et al.* [4].

The Boltzmann machine is a fully connected network with symmetric bidirectional connections. In addition, certain units of the network are designated as *input units,* certain other units are designated as *output units,* while the remainder are the *hidden units.* This division of the network is entirely arbitrary, since the units are indistinguishable.

The units in the network can be in one of two states, which may be denoted as *on* and *off,* 0 and 1, or 1 and -1. The network has an energy function of the same form as that of the Hopfield network:

$$E(t) = -\frac{1}{2}\sum_i \sum_{j\neq i} w_{ij}\mu_i(t)\mu_j(t) + \sum_i \mu_i(t)\theta_i$$

where $\mu_i(t)$ is the state of the ith unit at time t, w_{ij} is the weight on the link between the ith and jth units, and θ_i is the threshold of the ith unit.

The network also has a control parameter T, the temperature parameter, which controls the simulated annealing process when the network is run.

The figure shows a simple Boltzmann machine.

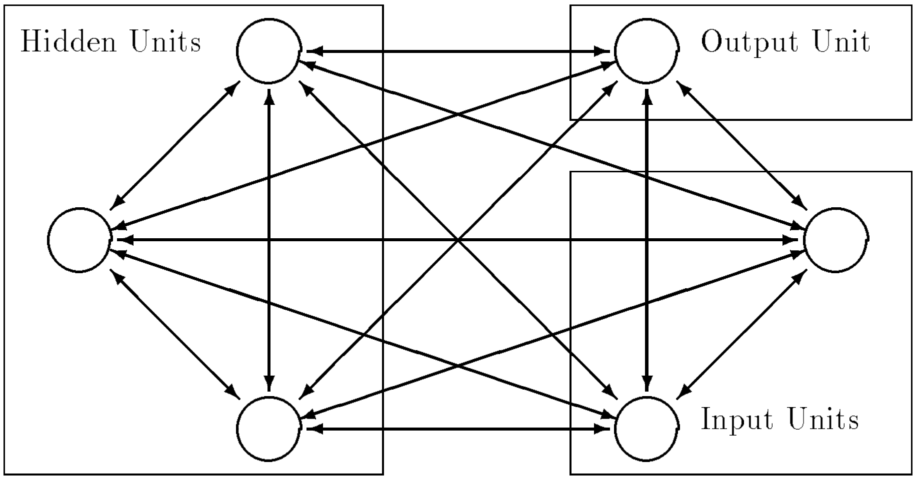

Figure 5.19: A simple Boltzmann machine

Network Operation

The determination of the network weights of the Boltzmann machine is a two-step process. In the first step, the states of the input and output units are held fixed and the states of the hidden units are modified by the simulated annealing process until a steady state is achieved. When this happens, statistics on the frequency with which pairs of units are both *on* are collected. This process is repeated for all the exemplar pairs several times. In the second step, only the states of the input units are held fixed, and the states of the hidden units and the output units are modified by the simulated annealing process until a steady state is reached. When this happens, statistics on the frequency with which pairs of units are both *on* are again collected. The statistics collected are used to update the network weights and the procedure is repeated until the weights stabilise.

Suppose the network has N_i input units, denoted $\mu_1, \ldots, \mu_{N_i}$, N_h hidden units, $\mu_{N_i+1}, \ldots, \mu_{N_i+N_h}$, and N_o output units, $\mu_{N_i+N_h+1}, \ldots, \mu_{N_i+N_h+N_o}$. Put $N = N_i + N_h + N_o$. The weights w_{ij} and thresholds $\theta_i, 1 \leq i, j \leq N$, are initialized to some arbitrary values, subject to $w_{ij} = w_{ji}, i \neq j$, and $w_{ii} = 0$.

The following procedure is repeated until the weights stabilise.

The input and output pairs are selected in some order for presentation to the network. The states of the input and output units are fixed at the values determined by each selected input and output pair, the states of the hidden units are set to some random initial values, and the temperature is set to some high value.

Simulated annealing is used to bring the network to a steady state. The hidden units are chosen in some random order and the quantity $\Delta E = \sum_j w_{ij}\mu_j(t) - \theta_i$ is computed, where the chosen unit is the ith one, and $\mu_j(t)$ is the state of the jth unit at time t. A random number between 0 and 1, denoted ρ, is also chosen. The state of the chosen unit is udpated according to the following rules:

- if $\Delta E < 0$ the state of the unit is changed;
- if $\Delta E \geq 0$ and $\rho < \exp(-\Delta E/T)$ the state of the unit is changed;
- otherwise the state of the unit is not changed.

After cycling through the units a number of times, the temperature parameter is reduced and the procedure is repeated. When the annealing schedule is completed, the network is run for a number of cycles while the states of pairs of units are examined, and the pairs which are both in the *on* state are recorded.

After all the input and output pairs have been presented a number of times, the recorded results from each presentation are used to compute p_{ij}, the proportion of the time that both the ith and jth units were *on*.

The exemplar patterns are now presented to the network again, but this time only the states of the input units are held fixed, while the states of both the hidden units and the output units are changed according to the simulated annealing process. Statistics are collected for the computation of p'_{ij}, the proportion of the time that both the ith and jth units were on when the output units were free.

The weights are now updated. If $p_{ij} - p'_{ij} > 0, w_{ij}$ is incremented by a fixed amount. If $p_{ij} - p'_{ij} < 0, w_{ij}$ is decremented by a fixed amount.

Once the weights stabilize, the network can be used for recall. The states of the input units are fixed at the values determined by the selected input vector and the simulated annealing procedure is applied to the hidden units and the output units. When equilibrium is reached, the output vector can be read off the output units.

Theory of the Network

The theory of the network is based on statistical mechanical concepts. In particular, it can be shown that when the network is allowed to run according to the simulated annealing procedure, the states of the network occur with a fequency that depends on the Boltzmann distribution. This distribution is a fundamental one in statistical mechanics. The distribution is determined by a single parameter, which is the temperature in thermodynamical situations, and has the property that states with low energy are unlikely to occur when the temperature parameter is large, and much more likely to occur when the temperature parameter is small.

The Boltzmann distribution provides the rationale for the simulated annealing procedure. Starting the procedure at high temperature makes the network more likely to occupy high energy states. As the temperature is reduced, the low energy states become more likely. Very low temperatures favour very low energy states, which should be close to the global minimum.

The Boltzmann distribution also provides the rationale for the procedure used to update the weights. It can be shown that the quantity $p_{ij} - p'_{ij}$, described above, is an indicator of the direction and amount of the change in w_{ij} that will result in a better matching of the input and output patterns.

Applications

Ackley *et al.* [4] applied the Boltzmann machine to the **Encoder Problem**, which involves an attempt to create an efficient description of the input vectors by encoding them in the states of the hidden units. It also can be applied to the study of the dynamics of spin glasses.

The Boltzmann machine was among the architectures applied to the recognition of handwritten digits in [3].

An application of the Boltzmann machine to job-shop scheduling is presented in [5] and [6].

The principal limitation the Boltzmann machine is the length of time it takes to produce its results, both in training the weights and in recall. In addition, in spite of the use of simulated annealing, it can get stuck in local minima and fail to find the global minimum.

5.6.6 Bidirectional Associative Memory

Introduction

Bidirectional Associative Memory (BAM) is an extension of the Hopfield network introduced by Kosko [7]. The principal difference between BAM and the Hopfield network is that the units are divided into two sets and there are no connections between the units in each set. The BAM is described as being capable of learning pattern pairs, but this is a misleading description of its operation; it simply recalls stored patterns, which have been previously been divided into two parts by the network designer.

Network Characteristics

The BAM is not a fully connected network. Instead, the units are divided into two sets, or layers. The layers are generally designated as the A and B layers; the number of units in the A layer need not be the same as the number of units in the B layer. Each unit in each layer is connected to all the units in the other layer and to none of the units in the layer to which it belongs. The links are bidirectional.

The units in the network can be in one of two states, which may be denoted as *on* and *off,* 0 and 1, or 1 and -1. If the network has N_A units in the A layer and N_B units in the B layer, we will denote the state of the ith unit in the A layer at time t by $\mu_i(t)$ and the state of the jth unit in the B layer at time t by $\nu_j(t)$. The weight on the link joining the ith unit in the A layer to the jth unit in the B layer is w_{ij}, where $1 \leq i \leq N_A$ and $1 \leq j \leq N_B$. It may not be the case that $w_{ij} = w_{ji}$; in fact, since N_A need not be the same as N_B, in some cases both w_{ij} and w_{ji} may not be defined.

The network has an energy function whose form is similar to that of the Hopfield network:

$$E(t) = -\sum_{i=1}^{N_A}\sum_{j=1}^{N_B} w_{ij}\mu_i(t)\nu_j(t).$$

The figure shows a simple BAM.

Network Operation

As in the case of the Hopfield network, the weights are determined from a set of exemplar pattern pairs. The weights are computed once.

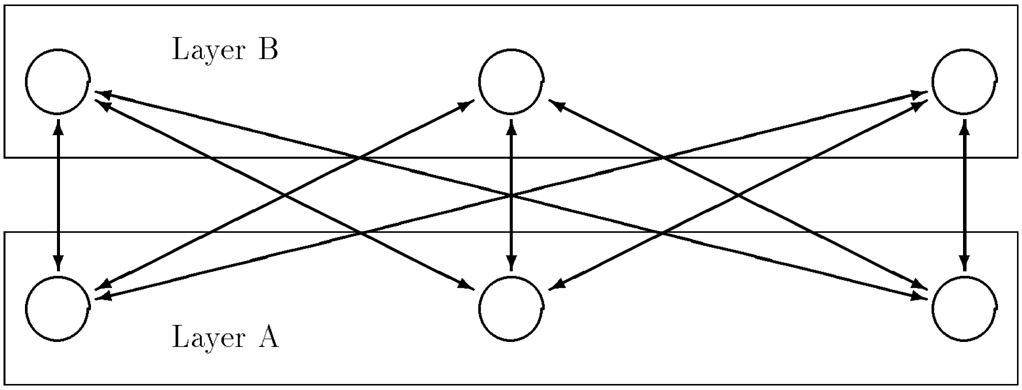

Figure 5.20: A simple Bidirectional Associative Memory

To run the network, the states of the units in the A layer are set according to some chosen pattern. The states of the units in the B layer are then determined for the states of the units in the A layer and the weights. New states for the units in the A layer are then computed from the states of the units in the B layer and the weights. This cycle between the layers is repeated until a steady state is reached.

The Network Equations

The weights are determined from a set of exemplar pattern pairs. If there are P pairs, $(\mathbf{a}_1, \mathbf{b}_1), (\mathbf{a}_2, \mathbf{b}_2), \ldots, (\mathbf{a}_P, \mathbf{b}_P)$, where $\mathbf{a}_k = (a_k^1, \ldots, a_k^{N_A})$ and $\mathbf{b}_i = (b_k^1, \ldots, b_k^{N_B})$ and $a_k^i = \pm 1, b_k^j = \pm 1$ for $1 \leq i \leq N_A, 1 \leq j \leq N_B, 1 \leq k \leq P$, then the weights are given by $w_{ij} = \sum_{k=1}^{P} a_k^i b_k^j, 1 \leq i \leq N_A, 1 \leq j \leq N_B$.

Let the initial states of the units in the A layer be set to $\mu_i(0)$ for $1 \leq i \leq N_A$. At time $t > 0$ we compute new states for the units in the B layer by computing the activations $\beta_j(t) = \sum_{i=1}^{N_A} w_{ij}\mu_i(t-1)$

and then setting the new states of the units to $\nu_j(t) = 1$ if $\beta_j(t) > 0$, $\nu_j(t) = \nu_j(t-1)$ if $\beta_j(t) = 0$ and $\nu_j(t) = -1$ if $\beta_j(t) < 0$, $1 \leq j \leq N_B$.

We then compute new states for the units in the A layer by computing the activations $\alpha_i(t) = \sum_{j=1}^{N_B} w_{ij}\nu_j(t)$ and then setting the new states of the units to $\mu_i(t) = 1$ if $\alpha_i(t) > 0$, $\mu_i(t) = \mu_i(t-1)$ if $\alpha_i(t) = 0$, and $\mu_i(t) = -1$ if $\alpha_i(t) < 0$, $1 \leq i \leq N_A$.

New states for the units in each layer are computed in alternation until a steady state is reached, which represents the output of the network.

Theory of the Network

The theory of the network is similar to that of the Hopfield network. It can be shown that each cycle reduces the value of the energy function, so that the network tends to a local minimum of the energy function. The determination of the weights ensures that the exemplar patterns are located at these minima, provided the number of patterns is small compared to N_A and N_B.

Applications

The poor storage capacity of the BAM architecture has limited its applicability.

There have been extensions of the architecture to cases where the states are continuously variable and where the weights are adapted over time instead of being fixed by the set of exemplar patterns.

5.6.7 ART

Introduction

Adaptive Resonance Theory (ART) was developed by Stephen Grossberg in his studies of the behaviour of models of systems of neurons. Grossberg has published a large number of papers, in which he develops the mathematical theory of such models, usually in terms of systems of differential equations, and applies this to actual neural systems. Like Kohonen, he has been particularly interested in systems that are capable of organizing themselves.

Grossberg has used his mathematical results to develop two neural network architectures, which implement nearest neighbour pattern classification. The first architecture, usually referred to as **ART1**, works on patterns of binary data. The second, **ART2**, is an extension of ART1 to arbitrary patterns. ART1 is described in [8], ART2 in [9] and [10].

Network Characteristics

Both the ART architectures are two-layer networks in which units in the first, or input, layer receive inputs of the data and feedback from the second layer. Units in the second, or output, layer receive inputs from the input layer and interact among themselves.

The ART networks implement a nearest neighbour classification algorithm whose simplicity is obscured by the complexity of the network architecture.

The network stores patterns as sets of weights assigned to the paths connecting the input units to each of the output units. Presentation of an input vector causes the output units to be activated, the amount of activation depending on the similarity between the input vector and the stored pattern. Finding the nearest neighbour is simply a matter of deciding which output unit has been activated the most.

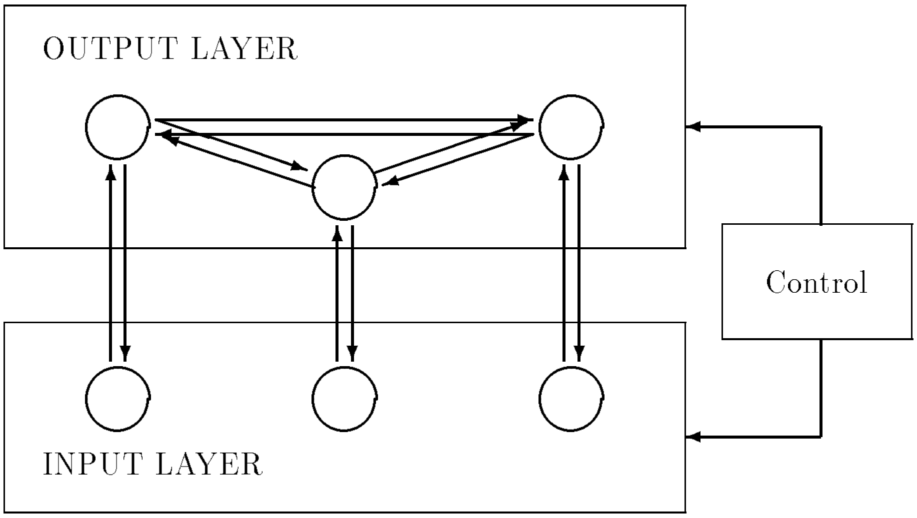

Figure 5.21: A simple ART1 network

The networks also have a mechanism for adding new units to the output layer. This mechanism is invoked whenever an input is presented which is dissimilar to all the existing stored patterns. The invocation of this mechanism is regulated by a control parameter. In fact, the control of the operation of the network is a complex process requiring interactions between the units in both layers and a control unit.

The figure shows a simple ART1 network.

Network Operation

The heart of the ART networks is a scheme for interaction between the units in the output layer which guarantees that the most active unit will suppress the activity of all the other units. After the output units have been activated, they are allowed to interact until only one remains active. This unit then activates the units in the first layer via another set of paths with their own weights. The input vector is then compared with the pattern that has been passed down from the output layer.

The comparison between these patterns is controlled by a parameter called the *vigilance threshold*, which determines the degree of similarity between patterns that are given the same classification. If the difference between the input pattern and the pattern that is passed down from the output layer is less than the vigilance threshold, then the active output unit is accepted as determining the correct classification of the input, and the weights associated with that unit are adjusted according to one of the schemes described below.

If the difference between the input pattern and the pattern passed down from the output layer is greater than the vigilance threshold, the activity of the active unit is completely suppressed, and process described above is repeated until either a classification is achieved, or there are no units left, in which case a new class is created by the addition of a unit to the output layer.

The weights that connect each output unit to the input units represent the pattern typical of the class to which the output unit belongs. The weights that connect the input units to each output unit represent the same pattern, except that their values are normalized.

The effect of the operation is as follows: for each input vector, the network finds the closest pattern among the output units. If this is within the distance determined by the vigilance threshold, this is accepted as the correct classification and the weights adjusted to make the stored pattern more similar to the input vector. Otherwise, the network continues to go through the remaining output units in order of similarity to the input vector. If any of them is found to be within the distance determined by the vigilance threshold, it is chosen as the correct classification. If no output pattern is sufficiently similar, a new output unit is created.

Both ART1 and ART2 operate in this way. The only difference between the two is that ART1 inputs consist of binary patterns, while the inputs to ART2 networks can have arbitrary values, which are normalized by additional operations that are applied within the input units.

Theory of the Network

The equations that are used to determine the states of the units are based on the results of Grossberg's research into systems of differential equations. The principal theoretical result underpinning the operation of the ART networks is the *Cohen-Grossberg Stability Theorem.* This is a very general and technical result about a class of ordinary differential equations. In the context of ART, it guarantees that the units in the output layer will reach a steady state and not oscillate periodically or chaotically.

Applications

Grossberg and his collaborators have applied the ART architectures to many problems, including modelling of human behaviours [11] and classification of radar images [12].

The Defense Science and Technology Organization in Adelaide has applied ART1 to the classification of infra-red ship images [13].

5.6.8 Neocognitron

Introduction

The Neocognitron was developed by Fukushima [14] in an attempt to construct a neural network architecture based explicitly on knowledge about real brains. It is probably the most complex neural network architecture developed to date, and probably the most limited in terms of the scope of its possible applications.

The Neocognitron was designed to recognize handwritten digits. Much of its complexity stems from an attempt to make the recognition process robust against variations in different people's handwriting and against variations in the position of the digit presented to it.

The design of the Neocognitron was inspired by knowledge of the visual system derived from neurophysiological experiments such as those of Hubel and Wiesel ([15] and [16]). These experiments have discovered a great deal about the structure and function of the parts of the brain that derive their inputs primarily from the retina. In particular, it has been discovered that there is only a small number of types of cells receiving inputs directly from the retina and these cells have very restricted functions; that the local relationships between cells in the retina are preserved in the organization of the neural pathways; and that the visual system appears to be structured hierarchically, with information being passed from one level of the structure to the next. These discoveries are reflected in the design of the Neocognitron.

There are a number of features that make the Neocognitron different from other neural network architectures. These include its *hierarchical structure,* its *training methodology,* and its *network dynamics*. These features will be described in greater detail below.

The hierarchical structure of the Neocognitron is perhaps the most significant difference between it and other network architectures. It makes it possible to piece together the global picture in a number of steps, so that the amount of processing to be done by any one layer is limited. In addition, each of the units receives input from only a small fraction of the units in the previous layer, reducing the number of connections between units.

Network Structure

The Neocognitron is designed as a system to recognize the handwritten digits from 0 to 9. It has a hierarchical structure, being divided into nine layers.

The first layer is the input layer and is designated U_0. It consists of an array of photoreceptor units arranged in a 19×19 square. The remaining layers are divided into four sets of two layers each. Each of these sets has a U_S layer and a U_C layer, which are given numbers to indicate the set to which they belong. The flow of information is along the path $U_0 \rightarrow U_{S1} \rightarrow U_{C1} \rightarrow U_{S2} \rightarrow U_{C2} \rightarrow U_{S3} \rightarrow U_{C3} \rightarrow U_{S4} \rightarrow U_{C4}$.

The U_{S1} layer has $19 \times 19 \times 12$ units. Each unit in the U_0 layer has 12 units corresponding to it in the U_{S1} layer. Each of these units receives inputs from the corresponding units in the U_0 layer and from the eight units surrounding it. Each of these twelve units is trained to respond to a pattern that represents a small portion of a line segment. In this way, the $S1$ units function as a collection of line segment detectors spread over the entire input array.

The U_{C1} layer consists of $11 \times 11 \times 8$ units. The 12 bit patterns recognized by the $S1$ units actually correspond to 8 possible orientations of line segments, so outputs from equivalent $S1$ units are fed to the same $C1$ units. Each $C1$ unit receives inputs from $S1$ units corresponding to a 5×5 square of receptor units. This reduces the dimensions of the layers from $19 \times 19 \times 12$ to $11 \times 11 \times 8$.

There are similar arrangements between the succeeding layers, so that the U_{S2} layer has $11 \times 11 \times 38$ units, the U_{C2} layer has $7 \times 7 \times 22$ units, the U_{S3} layer has $7 \times 7 \times 32$, the U_{C3} layer has $7 \times 7 \times 30$, the U_{S4} layer has $3 \times 3 \times 16$, and the U_{C4} layer has ten units, which are used to indicate which of the digits has been presented to the network. The gradual reduction in the first two dimensions of each layer reflects the gradual increase in the proportion of the input array which affects each unit. The alternating reduction and increase in the size of the third dimension reflects encoding of a number of features in the output from the previous layer and the subsequent reduction of this number by identifying features that are equivalent for the purposes of digit recognition.

The units described above are all excitatory units. In addition, there are cells between each of the U_C layers and the following U_S layer which have an inhibitory effect on the units in the following layer. A little arithmetic shows that there are more than 50000 units in the Neocognitron.

The Neocognitron is a feedforward network with the outputs from each layer (except the last) becoming the inputs to the next layer. It is not a fully connected network; there are no connections between units belonging to the same layer, and each unit is connected to some, rather than all, of the units in the previous layer. This means that the number of connections is less than that of a Backpropagation network of similar size and far less than that of a comparable Hopfield network. Even so, the Neocognitron has about 14 million connections.

The Network Equations

The output of each unit at any time is represented by a non-negative real number. The connections between units have weights associated with them which modify the inputs from the previous layer.

The equations describing the operation of the units are complex, and the description which follows is condensed and simplified.

Each unit in each of the U_S layers takes the outputs from the excitatory units in the previous layer to which it is connected, multiplies them by the appropriate weights, and sums the products. We will denote this sum by e. It also has an input from an inhibitory unit, which is multiplied by its own weight. This product is denoted by h. The effects of the excitation and inhibition are combined by computing $x = ((1+e)/(1+h)) - 1$. The output of the unit is 0 if $x < 0$ and x otherwise. The output will be equal to e if $h = 0$ and 0 if $h > e$.

Each unit in each of the U_C layers takes the outputs from the excitatory units in the previous layer to which it is connected, multiplies them by the appropriate weights, and sums the products. We will denote this sum by x. The output of the unit is $x/(\alpha_l + x)$ is $x \geq 0$ and 0 otherwise. α_l is a parameter associated with the lth layer which determines the degree of saturation of the output of all the units in that layer.

The inter-layer inhibitory units each send an output to a single unit in the following U_S unit and receive inputs from the same units in the preceeding U_C layer as the cell which receives its output. The output of each inhibitory unit is the weighted root mean square of its inputs.

Training the Network

The network is trained one layer at a time. For each layer, a set of training patterns is defined, and the desired responses of the units in the layer to those patterns is determined. The patterns are presented one at a time, and the weights on the connections to the units in the layer are adjusted until the units respond as required.

The training procedure differs from that of other networks in two ways. First, as mentioned above, the network is trained a layer at a time. The training is supervised, and directed towards having the units in each layer behaving in a predetermined way to each input pattern. This requires that the behaviour of all the units in all the layers be specified in advance, and that the network can only work effectively on the class of patterns on which it was trained. Training for the digits will require a different procedure from training for ten alphabetic characters.

Second, the training is replicated across the input array. This is most easily explained in relation to the U_{S1} layer. There are 12 units in this layer that correspond to each unit in the input (U_0) layer. The intention of the training process is that the twelve units at each position respond to the same set of input features. To ensure this, changes to the weights are determined for a single position in the input array, say at (x_0, y_0), and applied to the weights at all positions in the next layer, namely at (x, y, z_0), for all x and y. This method ensures that the network is insensitive to variations in the positions of the input patterns.

Once the network is trained, presentation of an input pattern should result in one of the units in the U_{C4} layer having the largest output value. This unit indicates which of the digits was presented as input.

Applications

The Neocognitron was designed as a system for recognizing handwritten digits. It may be used for other pattern recognition tasks by training it on an appropriate data set. However, the network is so complex and requires so much specialized training and tuning that it is unlikely to have many applications.

5.6.9 References

The following papers were cited above.

[1] Hopfield, J.J. "Neural networks and physical systems with emergent collective computational abilities", *Proceedings of the National Academy of Sciences,* **79**, pp. 2554–2558, 1982.

[2] Hopfield, J.J. "Neurons with graded response have collective computational properties like those of two-state neurons", *Proceedings of the National Academy of Sciences,* **81**, pp. 3088–3092, 1984.

[3] Pawlicki, T.F., Lee, D.-S., Hull, J.J., and Sihari, S.N. "Neural network models and their application to handwritten digit recognition", *Proceedings of the IEEE International Conference on Neural Networks,* vol. II, 1988, pp. 63–70.

[4] Ackley, D.H., Hinton, G.E., and Sejnowski, T.J. "A learning algorithm for Boltzmann machines", *Cognitive Science,* **9**, pp. 147–169, 1985.

[5] Foo, Y.-P. S., and Takefuji, Y. "Stochastic Neural Networks for Solving Job-Shop Scheduling: Part 1. Problem Representation", *Proceedings of the IEEE International Conference on Neural Networks,* vol. II, 1988, pp. 275–282.

[6] Foo, Y.-P. S., and Takefuji, Y. "Stochastic Neural Networks for Solving Job-Shop Scheduling: Part 2. Architecture and Simulations", *Proceedings of the IEEE International Conference on Neural Networks,* vol. II, 1988, pp. 283–290.

[7] Kosko, B. "Bidirectional associative memories", *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC–18**, pp. 42–60, 1988.

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

[9] Carpenter, G. and Grossberg, S. "ART2: Self-organization of stable category recognition codes for analog input patterns", *Proceedings of the IEEE First International Conference on Neural Networks,* vol. II, 1987, pp. 727–736.

[10] Carpenter, G. and Grossberg, S. "ART2: Self-organization of stable category recognition codes for analog input patterns", *Applied Optics,* **26**, pp. 4919–4930, 1987.

[11] Grossberg, S. and Mingolla, E. "Neural dynamics of perceptual grouping: Textures, boundaries and emergent segmentations", *Perception and Psychophysics,* **38**, pp. 141–171, 1985.

[12] Carpenter, G. and Grossberg, S. "Invariant pattern recognition and recall by an attentive self-organizing ART architecture in a nonstationary world", *Proceedings of the IEEE First International Conference on Neural Networks,* vol. II, 1987, pp. 737–746.

[13] Lozo, P., Johnson, R.P., Nandagopal, D., Nussey, G., and Zyweck, T. "An Adaptive resonance Theory (ART) based Neural Network Approach to Classifying Targets in Infrared Images", *Proceedings of the*

Second Australian Conference on Neural Networks, 1991, pp. 22–25.

[14] Fukushima, K., Miyake, S., and Ito, T. "Neocognitron: a neural network model for a mechanism of visual pattern recognition", *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC–13**, pp. 826–834, 1983.

[15] Hubel, D.H., and Wiesel, T.N. "Receptive fields, binocular interaction and functional architecture in cat's visual cortex", *Journal of Physiology,* **160**, pp. 106–154, 1962.

[16] Hubel, D.H., and Wiesel, T.N. "Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat", *Journal of Neurophysiology,* **28**, pp. 229–289, 1965.

The following books were also used in preparing these sections.

Anderson, J. A., and Rosenfield, E. *Neurocomputing: Foundations of Research,* The MIT Press (Cambridge, Mass.), 1989.

Beale,R., and Jackson, T. *Neural Computing: An Introduction,* Adam Hilger (Bristol), 1990.

Hecht-Nielsen, R. *Neurocomputing,* Addison-Wesley (Reading, Mass.), 1991.

Simpson, P.K. *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations,* Pergamon Press (New York), 1990.

⋆[16]

[16]That's all from Chris. You're back with me again from now on.

5.6.10 Quadratic Neural Nets: issues

The MLP cuts up the space of a finite binary data set into pieces by hyperplanes. This allows us to fit functions from a family so as to get agreement on the function sample defined by the data set, and the algorithmic process consists of shuffling hyperplanes about a space until they cut the data up in a satisfactory way.

If you were to be given a two dimensional problem with data consisting of noughts and crosses or black circles and white circles and invited to place a collection of hyperplanes judiciously so as to separate tyhe classes, you could probably do it faster than a computer could do it using a neural net. For some easy problems such as the double spiral data set you could do it in a few seconds, while it is likely to take mini-supercomputers a few weeks using Back-Propagation for three layer nets. Even given the remarkable powers of the eye-brain combination, this does not inspire much confidence in Back-Prop.

Some reflection on the reasons why Back-Prop is so bad, or a little sensible experimenting, lead you to see that you are hoping that a collection of hyperplanes will fall into a nice set of positions, each complementing the others. Why should they? There is nothing at all in Back-Prop to make them. Fahlman pointed this out (see the references), and suggested more thoughtful ways of going about the job of partitioning data than taking random walks in a huge state space. Whether the resulting algorithm can reasonably be called a neural net algorithm is open to dispute, but it sure out-performs Back-Prop.

A little more reflection suggests that doing it with hyperplanes is pretty dumb anyway. If you are going to do it using the smallest number of units, you are likely to want to decompose your sets of points into a union of convex sets, as in the case of the articulated four layer net. Why not start off with convex sets and plonk them onto the data? Specht did this for spheres, why not use ellipses in two dimensions or hyperellipsoids in general? They are all specified by a positive definite quadratic form, or alternatively by a symmetric positive definite matrix. That's no big deal, since defining a simplex, the simplest closed convex shape you can make out of hyperplanes, in dimension n takes $n + 1$ hyperplanes, each of which takes n numbers. This is about twice the amount of information it takes to specify a quadratic form.

Which leads to a number of questions, one of which might be: what, if anything, does any of this have to do with neurons in people's heads? Are quadratic forms more plausible than affine discriminants? I shall argue later that the answer is 'yes'. The next question is, If you want to tell noughts from crosses in the double spiral problem, for instance, how do ellipses help you? The answer is that you have two species of ellipses, red ellipses for the noughts and green ones for the crosses. Then you attract the red ellipses in towards the nought data, the green one toward the cross points, and they totally ignore each other. If your rule of attraction is judiciously chosen, you wind up with the answer to the next question, how do you train the quadratic neurons? How can I attract them towards the data? I'll go into that a little later on, too. It requires a little thought.

The $Fig.$5.22 shows a partial solution to the double spiral problem which may be obtained in times four or more orders of magnitude faster than may be found using the classical piecewise affine neural nets, that is to say it's at least ten thousand times faster. The artist got bored of drawing vile shaped ellipses and so only put them on the O's, leaving the reader to draw some on the X's if he feels so inclined, after which a small child may be invited to colour them in, yielding an image much like one of those on disk.

A case may be made out, and indeed will be made, for the proposition that real, live, wetware neurons do something rather like fitting quadratic or higher order forms to data. The effect of this may be something not too different from an adaptive Probabilistic Neural Net, and it tends to suggest that neurons are actually making local estimates of the density of data. This is an important conception of the function of a neuron, so deserves some extended exegesis. This will be done in a later chapter.

For the present, it suffices to observe that if data comes in through

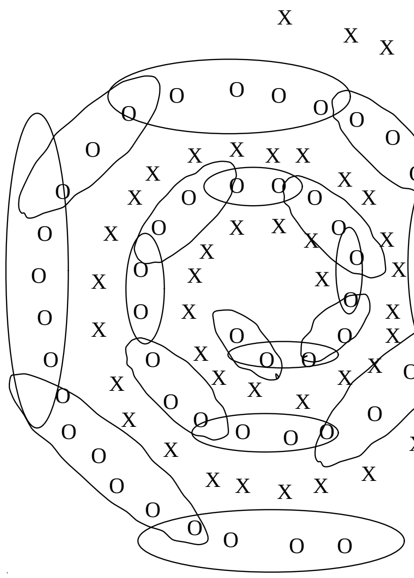

Figure 5.22: A partial solution to the Double Spiral Data Set.

sensors in a stream, extracting information may require an adaptive process somewhat different from usual statistical methods for computing model parameters. Moreover, although a gaussian *pdf* is determined by a quadratic form, a quadratic form may be used to define other *pdf*s or may be used as are hyperplanes simply as delimiters of different regimes.

This way of looking at things unifies some otherwise disparate matters. One satisfying feature of Quadratic neural nets is that enclosing points of different categories by hyperellipsoids rather than by hyperplanes makes Neural Net methods look a whole lot more like statistics and gaussian mixture modelling, as well as being many thousands of times faster on many problems. A second is that classifying different types of point by having different categories of model neuron responding to them, reduces classifying to multiple clustering. So function fitting and clustering have at least some common elements from a neural net point of view.

After I've told you how to make positive definite quadratic forms, otherwise hyperellipsoids, jump through hoops and find data points, you will never want to use Back Propagation with Multi-layer Perceptrons for classifying data ever again. This will save you a lot of computer time.

5.7 Summary of Chapter Five

We started the chapter with a brief and possibly prejudiced account of some of the history of neural nets. We raised the issue of the extent to which they could accomplish wonders beyond the capacity of statistical methods, or whether connectionists should be having a mid-life crisis. We continued by first studying the alpha perceptron or single unit neural net, and proceeded by choosing to investigate the geometry of committee nets (three layered neural nets, in the conventional usage) and then of articulated nets with an extra layer. This allowed us to gain some insight into the principles of chopping up, by hyperplanes, the measurement space for a pattern classification task. We also constructed some simple algorithms which extended the perceptron convergence algorithm for a single unit to the case of these three layer and four layer nets. The Back-Propagation algorithm could then be seen in perspective: it was easy to see what it did and how it did it, and to perceive its practical limitations. The Back-Prop algorithm was outlined; the details of the algorithm were brushed off in rather an off-hand manner with a referral to the literature on the grounds that (a) clever people like you and me could now work it out from first principles if that looked like a good idea and (b) it looked a pretty bad idea. A discursion into function fitting followed, again with the intention of casting doubts on the standard procedures involving multi-layer perceptrons for doing this job, and shaking any ingenuousness the reader might have about the value of the contribution of pure mathematicians, who are as desperate to publish anything that might be mistaken for an article of practical value as any of us, and a good deal worse equipped to recognise practical value if they should chance to meet it.

The next section considered the value of Feed Forward MLP's from the point of view of their effectiveness as models from the Rissanen data compression perspective. I argued that an MLP accomplishing a binary classification was using hyperplanes to store only the category of the data and said nothing about where the data is in the space. It is thus easy to compute the compression effected by a neural net of this type and compare it with the null model to see if it is actually extracting information about the disposition of the points of a category. We concluded that the number of data points must be more than nkP in order for the model to have any value, where n is the dimension of a committee net, k the number of units and P the precision, some value between 2 and 64 which was in principle data dependent. The situation was worse for more complex net topologies. This suggested a certain element of frivolity in some artificial neural net publications.

A brief survey of neural nets other than MLP's was conducted, the thrust being to expose the underlying ideas rather than to specify the algorithms in any great detail. It was considered sufficient to describe what the things actually did and outline how they did it, because this is often enough to put one off the whole idea of getting involved with them.

Finally, quadratic neural nets were introduced in order to show that they did what some of the others did, and did it more intelligently and consequently much faster. The case was weakened by omitting details which, it was promised, would emerge later.

5.8 Exercises

1. Construct a binary data set in the plane. Colour the points of one category red and the other green, and display them on a computer screen. Construct a three layer neural net for a two dimensional input, and display the lines representing the weights of the units in the hidden layer; a small arrow sticking out of the side of each plane should be added to indicate which is the

positive side. Lock all the last weights to the output unit at value 1. Program your machine to present data from the double spiral at random, use Back-Propagation to adapt the hidden unit weights, and display the (oriented) lines on the screen. Run it with varying numbers of hidden units and monitor the Square Error. Verify that the program is behaving itself by working with simple data sets first. Compare the times for convergence for data sets of varying complexity, and observe the behaviour of the net when there is no solution. Run your program on a double spiral data set.

2. With random initialisations of a set of gaussians, try solving the above data sets using the EM algorithm. Display the gaussians as ellipses using the methods of chapter four. Use red ellipses for the red data points, green ellipses for the green data points, and run them quite independently. Use a Bayesian method for determining the correct category; also use the Mahalanobis distance to the closest ellipse centre.

3. Increase the dimension progressively for both of the above systems. You will not be able to display the results graphically now, but keep records of times to convergence using a variety of random initialisations for both systems.

4. What conclusions do you draw from observing the performance of these two systems on different kinds of data set?

5.9 Bibliography

1. G.F. Simmons, *Introduction to Topology and Modern Analysis*, McGraw-Hill, 1963.

2. Teuvo Kohonen, *Self-organization and associative memory, 3rd Edn.* Berlin ; New York : Springer-Verlag, 1989.

3. Michael Alder, Roberto Togneri, Edmund Lai and Yianni Attikiouzel, *Kohonen's algorithm for the numerical parametrisation of manifolds* Pattern Recognition Letters 11(1990)pp313-319.

4. Donald Specht, *Probabilistic Neural Networks* Neural Networks 1990

5. M.D. Alder, R. Togneri and Y. Attikiouzel *Dimension of the Speech Space* IEE Proceedings-I Vol 138, No. 3, June 1991.

6. Sok Gek Lim, Michael Alder & Paul Hadingham *Adaptive Quadratic Neural Nets* Pattern Recognition Letters,13, May 1992, pp325-329.

7. T. Khanna, *Foundations of Neural Networks* Addison Wesley 1990.

8. Jacek Zurada *Introduction to artificial neural systems* St. Paul: West, 1992 .

9. Michael Alder, Sok Gek Lim, Paul Hadingham and Yianni Attikiouzel *Improving Neural Net Convergence* Pattern Recognition Letters 13 (1992)pp331-338.

10. S. Fahlman and C. Lebiere *The Cascade-correlation learning architecture* in D.S. Toretzky (Ed) *Advances in Neural Information Processing Systems 2* Morgan Kauffman, 1990.

11. R. Togneri, M.D. Alder and Y. Attikiouzel *Dimension and Structure of the Speech Space* IEE Proceedings-I Vol.139, No.2, April 1992.