University of Stirling
Department of Computing Science and Mathematics

Computing Science Technical Report

Learning XOR: exploring the space of a classic problem

"One could almost write a book about the process of solving XOR …" (McClelland and Rumelhart 1988, page 303)

Richard Bland

June 1998

Table of contents

1. Background

Boolean functions of two inputs are amongst the simplest of all functions, and the construction of simple neural networks that learn to compute such functions is one of the first topics discussed in accounts of neural computing. Of these functions, only two pose any difficulty: these are XOR and its complement. XOR occupies, therefore, a historic position. It has long been recognised that simple networks often have trouble in learning the function, and as a result their behaviour has been much discussed, and the ability to learn to compute XOR has been used as a test of variants of the standard algorithms. This report puts forward a framework for looking at the XOR problem, and, using that framework, shows that the nature of the problem has often been misunderstood.

The report falls into three main parts. Firstly, in Section 2 I review neural nets in general and the XOR problem in particular. This section is based on classic studies and the material will be familiar to most readers. Secondly, in Section 3 I look in considerable detail at the problem-space of the XOR problem using the simplest network that can solve the problem using only forward transmission with each layer communicating only with the next layer. Finally, in Section 4 I give a comprehensive account of the error surface. I show that this surface has exactly sixteen minima and that each is a solution to the XOR problem. This is a new result and corrects errors made by many previous authors.

2. Neural Nets and the XOR problem

a) Neural nets

Neural nets are built from computational *units* such as that shown in Figure 1. This unit is a rough analogue of the animal neuron, which connects to other neurons or biological devices at *synapses*: these connections are inputs except for a single output down the *axon* of the neuron. The sensitivity of the neuron to its inputs is variable. The output is all-or-nothing: the neuron either *fires* or produces no output.

The design of the corresponding computational unit follows this description fairly closely. The unit has a number of *inputs*: these may be from the outside world or from some other unit. Each input has an associated *weight*. Each unit is considered to receive a single input stimulus made up of the weighted sum of the inputs. Writing a_j as the *j*th input (or *activation*) and w_j as the corresponding weight, we write the summed input to the *i*th unit as

$$in_i = \sum_j w_{ji} a_j \tag{1}$$

The unit then applies a function *g* to the summed input. This function is typically a step function such as

$$step(x) = \begin{Bmatrix} 1, \text{if } x \geq t \\ 0, \text{otherwise} \end{Bmatrix} \tag{2}$$

where *t* is the *threshold* of the function. As we shall see, for many purposes it is important that the function is differentiable, and for this reason the sigmoid function

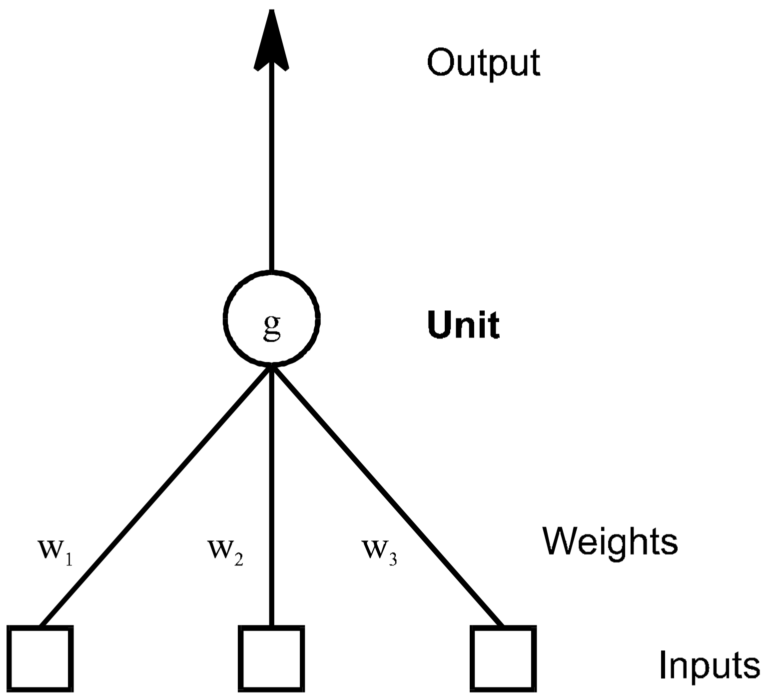

Figure 1: A Unit of a neural net

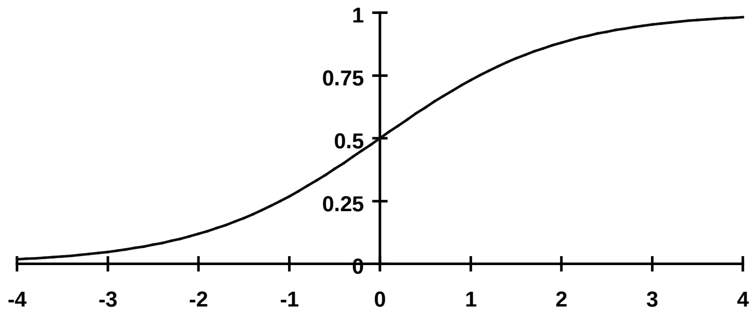

Figure 2: Sigmoid function

$$sigmoid(x) = \frac{1}{1+e^{-k(x-t)}} \tag{3}$$

is convenient. Here again the parameter t gives a threshold for the function: the function approaches 0.0 as $x \rightarrow -\infty$ and 1.0 as $x \rightarrow \infty$, and for $t = 0$ takes on the value 0.5 at $x = 0$. The parameter k affects the steepness of the transition between 0 and 1 in the centre of the curve. Figure 2 shows this function for $k = 1$ and $t = 0$. We shall use these values throughout this paper.

Applying the function g to the summed inputs gives us the *activation value* (the output) of the ith unit:

$$a_i = g_i(\sum_j w_{ji} a_j) \tag{4}$$

where the subscript on the g indicates that each unit may have its own function: typically, the functions differ only in the threshold values, t, and it is convenient to remove this difference by giving each unit an extra input whose value is always -1. The weight given to this input thus has an effect identical to that of the parameter t in Equation (3), and using this weight in preference to manipulating t is algorithmically simpler. (This extra input is sometimes referred to as the *bias unit*.)

These units can represent (some) Boolean functions. For example, if g is the step function of Equation (2), then the unit shown in Figure 3 computes the function X AND Y, where X and Y are the two variable inputs, $X, Y \in \{0,1\}$, with 0 representing *False* and 1 representing *True*. The bias unit is the right-hand input, which together with its weight gives the function g a threshold of 1.5. A more compact representation of this is given in Figure 4, where the threshold has been written inside the circle representing the unit.

These units may be combined into networks by assembling them together in *layers* of units in parallel, and/or by making the outputs of one unit serve as inputs to other units. A network in which there are no cycles is called a *feed-forward* network: we discuss only feed-forward networks in this report. Units whose outputs serve only as input to other units, and whose behaviour is not directly visible in the outside world, are said to be *hidden*. A feed-forward network containing no hidden units (which must therefore be single-layer) is called a *perceptron* (Minsky and Papert 1969). Thus Figure 1 is a representation of a single-unit perceptron.

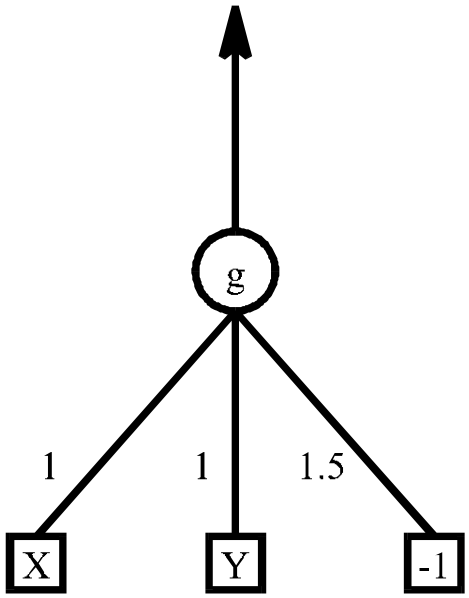

Figure 3: A unit that computes AND

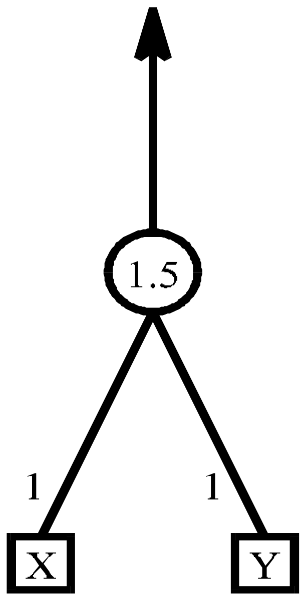

Figure 4: Alternative representation of AND

Perceptrons are remarkable because they can learn any of the functions that they can represent. From any starting set of weights, and given a set of examples of the inputs and the correct outputs (the *training examples*), there is an algorithm, the *perceptron learning rule* (Rosenblatt 1960), which adjusts the initial weights to a new configuration that represents the desired function. Provided that the function can be represented by a perceptron (we shall see what such functions look like in a moment) this algorithm always converges, given enough training examples. For any set of inputs and weights there will be a true output *T* and an actual output *O*. Then we alter each weight w_j using the following rule:

$$w_j \leftarrow w_j + \eta a_j (T - O) \tag{5}$$

where η is a fractional constant called the *learning rate*. As one would expect, this rule is applied by iterating round the set of examples until the weights converge.

b) Perceptrons and XOR

Unfortunately, perceptrons are limited in the functions that they can represent. As Minsky and Papert showed (Minsky and Papert 1969), only *linearly separable* functions can be represented. These are functions where a line (or, in the case of functions of more than two arguments, a plane) can be drawn in the space of the inputs to separate those inputs yielding one value from those yielding another. Thus in each of the cases of the truth tables for AND and OR, shown in Table 1 and Table 2 in a form that represents the Cartesian space of the inputs, we can draw a diagonal line across the table to separate the **T** entries from the **F** entries.

Hence AND and OR can be computed by single-unit perceptrons. This is of course not the case for the function XOR (Table 3) because there is no line that can be drawn across the table to separate the **1**s from the **0**s. XOR is thus not computable by a perceptron.

	1	**0**	**1**
Y	0	**0**	**0**
		0	1
		X	

Table 1: Truth table for AND

	1	**1**	**1**
Y	0	**0**	**1**
		0	1
		X	

Table 2: Truth table for OR

	1	**1**	**0**
Y	0	**0**	**1**
		0	1
		X	

Table 3: Truth Table for XOR

It is easy to see that this principle of linear separability follows from the equation that defines the single-unit perceptron, Equation (4). Taking the function g to be the step function, the equation shows that the output value will be 1 when

$$\sum_j w_j a_j \geq t \tag{6}$$

(taking the summation across only the original inputs, ignoring the bias input) and zero otherwise. Thus the values are partitioned by

$$\sum_j w_j a_j = t \tag{7}$$

which for two inputs is the equation of a straight line in the plane of the inputs a_j, and for more inputs is the equation of a plane in the space of the inputs.

Table 4 shows the 16 possible Boolean functions of two Boolean inputs. The first column gives a hexadecimal reference number to each of these functions. The numbers have been derived from the truth tables of each function, shown in the second column[1].

Inspecting these truth tables, we can see that all but two are linearly separable: functions f_6 and f_9, which are, respectively, XOR and its complement, are not. Thus these two functions alone are not computable by a single-unit perceptron.

[1]The numbers are purely for reference and the ordering has no significance. The algorithm for producing the numbers is to sum the cells of the truth table, taking the cell (1,1) as having the value 8, the cell (0,1) the value 4, the cell (1,0) the value 2, and the cell (0,0) the value 1. It turns out that writing the functions in this order we have the second eight functions as the complements of the first eight, reflected round the division between functions 7 and 8.

i	f_i	
0	00 00	False
1	00 10	$\overline{X}\,\overline{Y}$
2	00 01	$X\overline{Y}$
3	00 11	$\overline{Y}$
4	10 00	$\overline{X}Y$
5	10 10	$\overline{X}$
6	10 01	$\overline{X}Y + X\overline{Y}$ (XOR)
7	10 11	$\overline{X} + \overline{Y}$
8	01 00	XY (AND)
9	01 10	$XY + \overline{X}\,\overline{Y}$
a	01 01	X
b	01 11	$X + \overline{Y}$
c	11 00	Y
d	11 10	$\overline{X} + Y$
e	11 01	$X + Y$ (OR)
f	11 11	True

Table 4: The 16 Boolean functions of two inputs

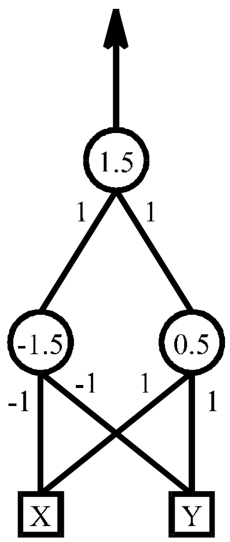

Figure 5: A net that computes XOR

<table>
<tr><td></td><td colspan="2">1.5</td><td></td></tr>
<tr><td></td><td>1</td><td>1</td><td></td></tr>
<tr><td colspan="2">-1.5</td><td colspan="2">0.5</td></tr>
<tr><td>-1</td><td>-1</td><td>1</td><td>1</td></tr>
</table>

Table 5: Tabular representation of Figure 5

<table>
<tr><td></td><td colspan="2">9</td><td></td></tr>
<tr><td></td><td>7</td><td>8</td><td></td></tr>
<tr><td colspan="2">3</td><td colspan="2">6</td></tr>
<tr><td>1</td><td>2</td><td>4</td><td>5</td></tr>
</table>

Table 6: Indices for vector representation of Table 5

These shortcomings of perceptrons can be overcome by allowing our network to have more than one layer, that is, to allow it to contain hidden units. In this way, we can build networks that will compute any function. The simplest form of this arrangement is that where each layer feeds only to the next, and a unit can only accept inputs from the layer immediately before it. Figure 5 shows such a network. There are two hidden units and a single output unit, and the output unit takes inputs only from the hidden units. The remainder of this report will look at networks with this topology.

In fact, with the weights shown in Figure 5, and using the step function, we have a network that computes XOR. Each of the three units computes a function from Table 4, the left-hand hidden unit (the one with the threshold of -1.5) computes f_7, $\overline{X}+\overline{Y}$, while the right-hand one computes function f_e, OR. Finally the output unit computes f_8, AND. A tabular form of Figure 5 is shown in Table 5. It should be clear that a bold box represents each of the three units, with the threshold weight shown above the two other weights. It is typographically most convenient to write the net simply as a vector of nine weights, and Table 6 shows the order in which we shall in future write these weights. Thus, the network of Figure 5 and Table 5 would be written as configuration (C1).

-1.0 -1.0 -1.5 1.0 1.0 0.5 1.0 1.0 1.5 (C1)

We have already seen that perceptrons can learn their weights using sample data. This is also true of nets of the form shown in Figure 5. Of course, the perceptron learning rule shown above in Equation (5), above, must be modified because for the hidden units we no longer have any direct estimates of the error (the difference between the 'true' output and that actually being produced by the unit). One conventional solution is *back-propagation* (Bryson and Ho 1969) in which the error detected at the output unit(s) is effectively distributed throughout the network.

As before, during learning we have for each output unit a true output *T* and an actual output *O*. For each output unit, we wish to update the weights relating to connections with the penultimate hidden layer. Considering the *i*th output unit and its connection with the *j*th unit in that layer, we use the rule

$$w_{ji} \leftarrow w_{ji} + \eta a_j g'(in_i)(T_i - O_i) \quad (8)$$

We see that this includes the derivative of the activation function *g* of the unit. Obviously, this means that the function must be differentiable, and it is for this reason that we use the sigmoid function, referred to earlier. Because this function's derivative is at its highest at zero, when the function value is half-way between the 'stable' outputs of zero and 1, we can say informally that we are making the greatest difference to the weight of units that are so far 'undecided'.

If we write the term $g'(in_i)(T_i - O_i)$ as δ_i then we obtain

$$w_{ji} \leftarrow w_{ji} + \eta a_j \delta_i \quad (9)$$

For the weights on the connections to the penultimate layer, that is from unit (or input) *k* to unit *j*, the update rule of (9) retains its form, becoming:

$$w_{kj} \leftarrow w_{kj} + \eta a_k \delta_j \quad (10)$$

but δ_j now alters: it is

$$\delta_j = g'(in_j)\sum_i w_{ji}\delta_i \qquad (11)$$

- and we see from this expression how the errors are indeed propagated backwards. This process continues until all the weights have been changed.

The process described so far is iterative: the set of training examples is used repeatedly until the weights settle. A possible refinement is to *batch* the changes to the weights: using this method the weights are not changed until after a complete set of training inputs have been applied. Instead, the changes are summed and applied to the weights at the end of the cycle. This makes a subtle difference to the behaviour of the algorithm on surfaces that are flat or nearly flat. If batching is not used, the configuration of weights can move in a closed cycle. If batching is used then the changes can cancel (summing to zero). It is important to realise that these are different phenomena, and would not usually happen for the same configuration. Also, if batching is not used, movement of the configuration after every training input may allow the algorithm to move away from saddlepoints.

It can be shown (Hertz, Krogh, and Palmer 1991) that the algorithm is of the familiar hill-descending type, which minimises the sum of the squared error terms (we shall call this the SSE). One variant of this procedure is to use a *momentum* factor: a proportion of the change last time is added to the change this time. This is introduced in an attempt to avoid local minima. Writing the change to the weight w_{ji} at time *t* as $\Delta_{ji}(t)$, then

$$\Delta_{ji}(t+1) = \alpha\Delta_{ji}(t) + \eta a_j \delta_i \qquad (12)$$

where α is a constant in the range 0 to 1.

As many authors have noted (see for example (Rumelhart, McClelland, and the PDP Research Group 1986) the back-propagation method will find solutions to the XOR problem. We turn now to a consideration of the space of XOR and of the behaviour of back-propagation in that space.

3. The space of XOR

a) Introduction

In this section, I present the results of an exploration of the space of the XOR problem: that is, I consider where the solutions are in the space of the weights. This discussion will focus exclusively on the behaviour of a network of the form shown in Figure 5: one with two hidden units, each of two inputs, and a single output unit. All the units will be considered to be using the same function *g*, modified by thresholds set by an additional "bias" unit whose input is always -1.

b) Boolean Class

We begin by considering the behaviour of such a network when the function *g* is the step function of Equation (2), and the inputs *X* and *Y* are two-valued. The two values are of course representing *True* and *False*, and we shall follow convention in giving these the values 0 and 1 respectively. Under these circumstances each of the three units computes one of the 16 functions of Table 4 (the list exhausts the possible Boolean functions of two Boolean inputs). We have seen that two of these functions are not computable by such (individual) units, so the units will be computing one of the 14 remaining functions. For any combination of three real values for the three weights of a unit, we can specify which function the unit computes. Thus, for example, if the threshold weight is *t* and the other two weights are w_x (the weight on the input *X*) and w_y (the weight on the input *Y*) and the following conditions are met:

$$t > 0,\ w_x \geq t\ ,\ w_y \geq t \qquad (13)$$

then the unit defined by these three values will compute f_e, OR. Thus, we could change the weights in the right-hand hidden unit of Figure 5 (the OR unit) to any values meeting the inequalities in (13) and the network would be "the same" — its constituent units would still compute the same functions as before and the network as a whole would still compute XOR. Thus, the networks of (C2) and (C3) are in this sense "the same" as the network of (C1).

-1.0	-1.0	-1.5	0.06	0.06	0.05	1.0	1.0	1.5	(C2)
-1.0	-1.0	-1.5	10.0	10.0	9.9	1.0	1.0	1.5	(C3)

Two lines of argument follow from this observation (if we remember that at this stage in the discussion we are still using the step function as the function *g*). The first is that we can show the form of a network like Figure 5 (and its corresponding Tables) as *8(7,e)* indicating that the output unit computes f_8 and this function is applied to the outputs of the hidden units computing f_7 and f_e. The order of writing the bracketed function numbers is significant, although as it happens in this example *8(e,7)* is also XOR because f_8 is symmetrical in its inputs. As a convenient label, let us call a form such as *8(7,e)* a *Boolean class* of networks. Thus, although each Boolean class contains infinitely many networks, they all are "the same" in the sense that their equivalent constituent units compute the same Boolean functions when using the step function.

Using this idea of the Boolean class, we can go on to consider what each Boolean class computes. There are 14^3 (2744) possible classes, each computing a single

Boolean function. The second column of Table 7 sets out the numbers of Boolean classes that compute each of the 16 possible Boolean functions of two inputs. (We shall discuss the third column of the Table later.)

Table 7 shows that 16 of the 14^3 possible classes compute f_6, XOR. These 16 classes are listed in the first column of Table 8. (We shall return to the other column of the Table in due course.) It can be seen that the classes occur in pairs in which the bracketed functions are reversed, such as the pair *1(1,8)* with *1(8,1)* or the pair *2(7,1)* with *4(1,7).*

c) Boolean Classes as regions in space

The second line of investigation pursues the idea that the various possible individual units can be though of as points in three-dimensional space — the space of the three weights. Further, the possible three-unit networks can be though of as points in nine-dimensional space. In terms of Boolean classes, the classes of units and of three-unit networks will be *regions* in three and nine-dimensional space respectively. Anticipating later material, these spaces and regions are (approximately) those that back-propagation learning has to navigate if a network is to learn XOR.

In considering the regions that classes of units occupy in three-dimensional space, I shall make use of a simplifying model. This model has the following characteristics:

1. The possible values for the three weights that define the unit are each randomly distributed in the interval $[-b..b]$, $b \in \Re^+$. The bound b can be thought of as the largest real number in use for weights in some particular computing device and as

Output function	Number of classes	Proportion of network weight-space
0	524	0.403447
1	144	0.017831
2	144	0.017578
3	128	0.016565
4	144	0.017578
5	128	0.016565
6	16	0.000760
7	144	0.009675
8	144	0.009675
9	16	0.000760
a	128	0.016565
b	144	0.017578
c	128	0.016565
d	144	0.017578
e	144	0.017831
f	524	0.403447
All	2744	1.000000

Table 7: Possible network outputs (using the step function)

Class	Volume
1(1,8)	4
1(8,1)	4
2(7,1)	4
4(1,7)	4
2(e,8)	4
4(8,e)	4
7(b,d)	4
7(d,b)	4
8(7,e)	2
8(e,7)	2
b(2,b)	8
d(b,2)	8
b(4,d)	8
d(d,4)	8
e(2,4)	8
e(4,2)	8

Table 8: Boolean classes computing XOR

such a bound certainly exists in practice its presence in the model seems very reasonable. (Of course, if the weights come from the domain of floating-point numbers rather than from the real numbers, then the distribution ceases to be strictly continuous, but this can be ignored.)

2. The probability distribution of each weight is uniform within the interval.
3. The weights are uncorrelated (i.e. the axes of the space are orthogonal).

The space of the weights of a unit can therefore be thought of as a cube of side *2b*. Inside this cube there are 14 regions corresponding to the 14 functions computable by a unit. Figure 6 gives five cross-sections through this cube: each is at right-angles to the dimension representing the threshold weight and thus shows an X-weight (x-axis) by Y-weight (y-axis) plane. The first cross-section is actually the top face of the cube, where $t = b$, the second cross-section (under the first) is one-quarter of the way down the cube, where $t = b/2$, and so on down to the last cross-section, the bottom face of the cube, where $t = -b$.

The regions of the space can (with some effort) be visualised from these cross-sections. Thus, the region where the unit computes f_8, AND, is an upside-down skew tetrahedron with its triangular base at the top right-hand corner of the upper surface of the cube and its apex at the origin. (This is an open boundary because the point at the origin, where all weights are zero, is not part of the region, belonging instead to the region of f_f, True.) Similarly the region of f_1 is an upside-down skew pyramid whose base is the third quadrant of the central plane and whose apex is the bottom left-hand corner of the cube. Its complement f_e is a skew pyramid whose base rests on the first quadrant of the central plane (as an open boundary: the quadrant itself belongs to f_f) and whose apex is at the top-right corner of the cube.

Going on from this, it is clearly possible with some further labour to calculate the volumes of the fourteen regions as fractions of the total space of the cube. The result of these calculations is shown in the last column of Table 9. This gives for each function's region the proportion of the whole space that it occupies: this is expressed in the apparently odd unit of the 48^{th} of the whole cube, chosen because this allows convenient entries in the table. Referring back to Table 8, we can see that six of the functions are not found in nets that compute XOR — these are functions 1, 3, 5, a, c, and f. From Table 9 we see that these account for 34/48 or 0.708 of the weight-space. Thus only just under 30% of the weight space of a unit calculates functions from which an XOR network can be built. Carrying these calculations forward into the nine-dimensional realm of such networks, we can calculate the proportions of the nine-dimensional hypercube occupied by each of the 2744 Boolean classes. This is easily done by multiplying together the figures from Table 9. For example, the class *1(1,8)* will occupy $(2 \times 2 \times 1)/48^3$ of the total volume of the hypercube. Hence, we can find the proportion of the total network weight-space occupied by each output function (given in the last column of Table 7). This Table shows that the classes that compute XOR occupy about 0.00076 of the total space of all such three-unit networks. We can also find the volume of each of the classes that compute XOR. These are given in the second column of Table 8.

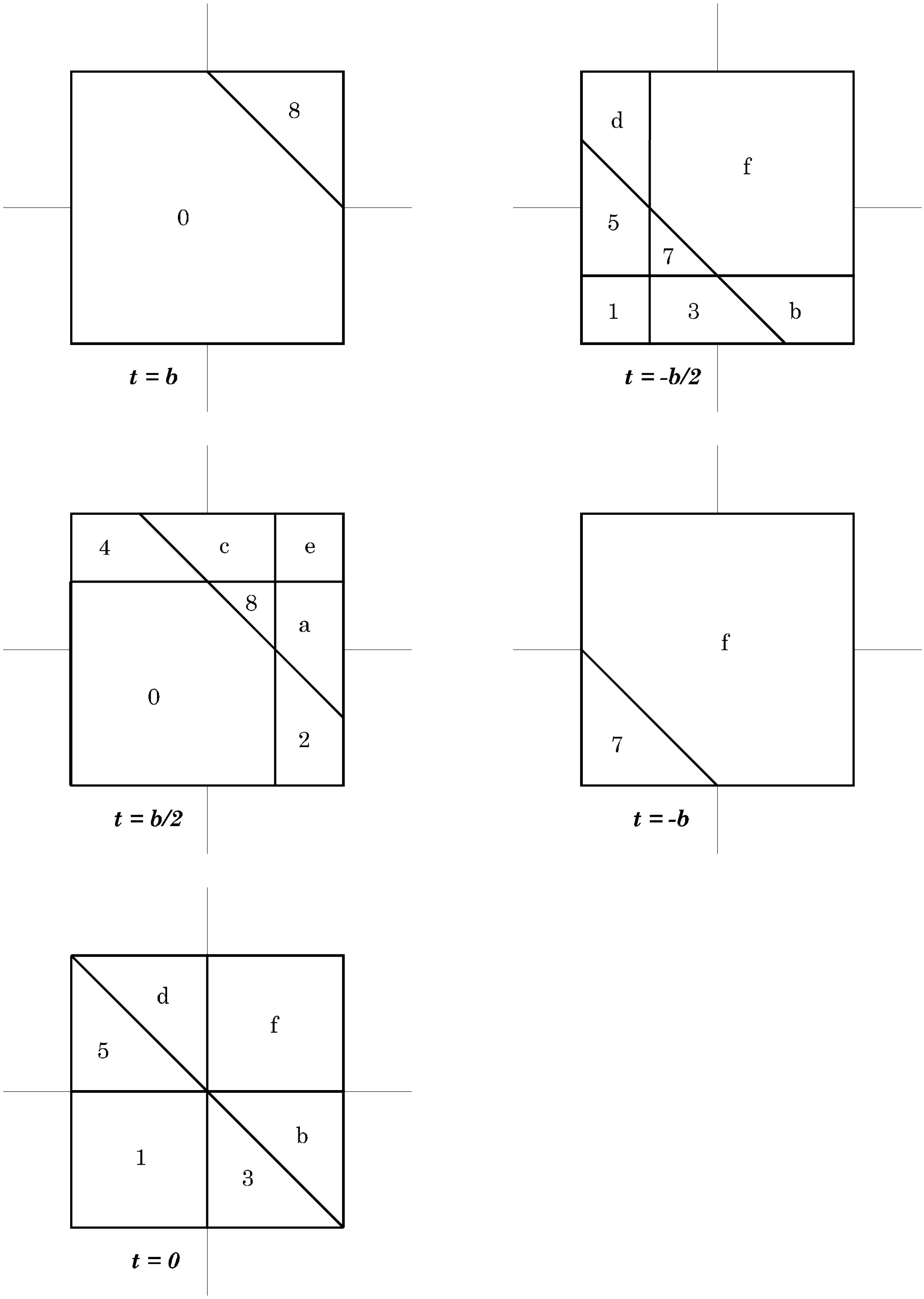

Figure 6: The space of the weights of a unit

The above discussion dealt with the situation when the step function of Equation (2) is employed. We found that using this function the space of the nine weights contains sixteen distinct regions corresponding to solutions to the XOR problem, and that together these regions made up about 0.00076 of the space.

i	f_i		**Size**
0	00 00	False	13
1	00 10	$\overline{XY}$	2
2	00 01	$X\overline{Y}$	2
3	00 11	$\overline{Y}$	2
4	10 00	$\overline{X}Y$	2
5	10 10	$\overline{X}$	2
6	10 01	$\overline{X}Y + X\overline{Y}$	-
7	10 11	$\overline{X} + \overline{Y}$	1
8	01 00	XY	1
9	01 10	$XY + \overline{XY}$	-
a	01 01	X	2
b	01 11	$X + \overline{Y}$	2
c	11 00	Y	2
d	11 10	$\overline{X} + Y$	2
e	11 01	$X + Y$	2
f	11 11	True	13
		Total	48

Table 9: Volumes of the 16 Boolean functions in the space of the weights of a single unit

d) Using the sigmoid function

Of course, in a practical situation where back-propagation learning is being employed we must use a differentiable function such as the sigmoid of Equation (3), a function cited in all standard texts: see, for example, (Hertz, Krogh, and Palmer 1991). This function can only achieve the values 0 and 1 in the limit. Hence, units using this function together with finite weights cannot produce exactly the values that we have agreed to call False and True, and so (strictly speaking) cannot compute any Boolean function. Of course, this is too strict: for practical purposes we must agree that approximations are acceptable. For example we could agree to treat values $< \varepsilon$ or $> (1-\varepsilon)$, where ε is some positive real quantity < 0.5, as being "really" False and True when they appear as the output of the output unit, while values in the range ε to $1-\varepsilon$ are unacceptable. Alternatively we can sum errors across the four cell of the truth table, and agree to accept a total error less than some small real value. The standard method is to sum the squared errors: in fact, it is this criterion that is minimised in the back-propagation gradient-descent method. Using such a method, a set of weights are taken to be computing some Boolean function when the sum of squared deviations from the true values in the function's truth table (the SSE) is smaller than some small positive real value ε, whose value might typically be taken to be 0.1. We shall follow this approach from now on. Thus there will potentially be two types of network: those that are said to be computing some Boolean function, because the truth table produced is acceptably close (by the sums-of-squared-errors criterion) to one of the sixteen standard truth tables, and those that are not computing any Boolean function at all, because their truth tables are too far from all of the sixteen standard tables. As a matter of nomenclature, we shall say that this second kind of net is computing a no-function.

Using the sigmoid, then, we might expect still to have sixteen distinct regions in the weight space, corresponding to those already described for the step function, where XOR is computed, but that the boundaries between these regions and their neighbours would be blurred. Where, using the step function, we had discontinuities in the space at the planes where the network jumped from computing one function to computing another function, we might expect to find transitional regions where no function is computed. For example, the network of Table 5, and (C1), which computes XOR when the step function is employed, in fact computes no function when the sigmoid is used. Its truth table, shown in Table 10, is obviously that of a no-function (and in fact has an SSE of 0.9951). However, the truth table produced by the net whose weights are those of (C1) multiplied by 100 (an identical net from the point of view of its Boolean class) produces XOR with an SSE that is zero to four significant figures. This second net, then, could serve as an example of a net clearly inside the XOR region, while the net of (C1) would be an example of one in the boundary of the region.

Y	1	**0.4366**	**0.4244**
	0	**0.4244**	**0.4366**
		0	1
		X	

Table 10: Truth table produced by the network of Figure 5 using the sigmoid

This intuitive reliance on an informal continuity argument would, then, lead us to expect that the solutions to the XOR problem would simply be those found for the step function, with each of the sixteen solution regions found earlier being separated from other regions by zones where the network is computing no-functions. The weight space would also contain regions where the network is computing the other 15 Boolean functions, AND, OR, and so on, and each of these regions will in turn be separated from other regions by zones of no-functions. The sixteen regions corresponding to XOR would occupy less of the weight space than before, because of the arrival of the intervening no-function regions, but we would hope that the back-propagation algorithm would still be able to find (at least one of) the solution regions using gradient descent. Looking at Figure 2, the graph of the sigmoid function, it is clear that small weights will tend to give no-functions because they will give small values of the summed input to the output unit (of course, this is precisely what led to the figures of Table 10). The behaviour of the net in this region would be of particular interest. We would expect that the surface produced by considering the SSE in the space of the weights (the “error surface”) would exhibit sixteen troughs, one for each of the sixteen solutions to XOR. These troughs will radiate from the origin, and have closed ends near the origin (where small weights lead to large error terms) and will become deeper and approach a height of zero as the weights become larger. The ridges between the troughs would be made up of parts of the weight-space where the net computes either no-function or some function other than XOR.

Unfortunately, however, matters are more complicated. Consider for example the truth table of Table 11. This has an SSE of 0.000071 as an approximation to XOR: clearly an excellent approximation. It was, however, produced by the network of (C4), using the sigmoid function. Now, that network is in Boolean class *2(7,0)* and thus does not compute XOR when using the step function: in fact, it computes f_7, $\overline{X}+\overline{Y}$.

-5.0	-5.0	-7.0	-5.0	-5.0	0.5	13.0	-37.0	6.0	*2(7,0)*	(C4)

Y	1	**0.9950**	**0.0046**
	0	**0.0009**	**0.9950**
		0	1
		X	

Table 11: Truth table produced by the network of (C4) using the sigmoid function

This counter-example shows that we cannot simply assume that the solutions using the sigmoid are essentially the same as the solutions already identified for the step function. It is clear that a number of questions must be answered in order to clarify the relationship between the sets of two solutions: for example, is the solution of (C4) drawn from a distinct new region of solutions, or is it a point on a descent into one of the regions that were identified for the step function? Or is it a point on a descent into “a local minimum”, a sort of second-class solution that we would want to discard? In which case, when is a solution not a solution? We address this methodological question in the next sub-section: the empirical questions will be addressed later.

e) Solutions, local minima, and saddlepoints: theoretical issues

The previous sub-section asked the question “when is a solution not a solution?” In this sub-section we look at a number of related methodological issues to do with solutions.

We know that using the sigmoid it is impossible with finite weights to reproduce XOR exactly: we also know that with physical computing equipment it is impossible to reproduce any real-valued function exactly. In many neural net applications we have no exact criteria that would enable us to recognise perfect performance on the part of a net (that is, we cannot specify exactly the function that the net is to compute). Thus for practical purposes we must accept that most solutions are approximations. In the present context, where we do know the function that we wish to compute, “a solution” is an outcome that is within a specified tolerance of an exact solution. In contexts where the target function cannot be specified exactly, “a solution” is one that leads to acceptable performance by the computer system (the robot picks up the brick 95% of the time, say). In both cases the decisions as to what outcomes are acceptable are guided, usually, by pragmatic considerations. When authors on neural nets speak pejoratively about “local minima” (as, for example, awkward areas to be ridden through using a large momentum term), they are, then, doing more than describing the mathematics of a surface. In such a context they mean that this particular minimum on the surface does not meet the current pragmatically decided criteria for a solution (and a better solution may be available). Using other criteria it would be “a solution.”

Therefore, the figures of Table 11 represent something that is a candidate for being a solution, and it would be hard to argue that it is not very close to an exact solution. For all practical purposes Table 11 is the truth table of XOR, and the network of (C4) is “a solution”.

A related question has to do with the meaning of the term “local minimum.” Some authors use this term very loosely, apparently meaning a portion of an error surface that

1. does not meet the criteria for a solution
2. back-propagation will not escape in some specified number of iterations

This usage is indefensible: firstly, it goes against the ordinary meaning of the words in mathematics, and secondly it depends on the parameters being used for back-propagation. Obviously, the maximum number of iterations is important, but so is the use of a momentum term. If a momentum term is used, then different results may be obtained if back-propagation is started in the region in question, or enters it later. A momentum term may be too low, or even too high (if the route out of the region is curved).

If we stay with the ordinary mathematical meaning of the term, then we obtain a more satisfactory definition. We wish to capture the idea that a local minimum is the least value in some region, but not the least value overall. (We shall use the term “global minimum” for the least value overall.) At the same time, we want to exclude the special case of a function that is decreasing at the edge of the specified region. For example, although it is true that $y = x^2$ has a minimum in the interval [2,3], and this minimum (the value 4) is greater than the global minimum (zero), we should not want to say that it had a local minimum in the interval. Finally, in connection with our present problem it is useful to be able to call a value a local minimum even if the value is approached asymptotically as some weight or weights become very large. Combining these various considerations, we arrive at the following: a local minimum *m* of a function $y = f(x_1,...,x_n), x_i \in \Re$ is a greatest lower bound *m* for *f* within some set of *n* open intervals $x_i \in (l_i, h_i)$, such that *m* is not a lower bound for *f* in the whole domain and *f* does not attain *m* for any possible $x_i = l_i$ or $x_i = h_i$. (Note that this final condition allows a local minimum to be approached as a limit as one or more of the x_i approach $\pm\infty$).

The obvious contrasts with a minimum are a maximum and a saddlepoint. The definition of maxima, if we were to need a definition, follows in an obvious way from our definition of minima. A saddlepoint is a point at which a tangent to the surface is flat (as it is at maxima and minima) but the immediate neighbourhood of the point contains some points at higher values and some at lower values. Thus, for example, the surface of $y = (x_1)^2 - (x_2)^2$ has a saddlepoint at the point $(x_1 = 0, x_2 = 0)$.

In the forthcoming material, we shall often show that particular points are not local minima. Normally in doing this, we shall employ an informal argument that will simply show that a path exists from the particular point, where all points on the path have the same or lower function values as the value at the original point. It is acknowledged that this *is* an informal approach and one that is open to abuse by an author prepared to act in bad faith. For example, using this method one could show that the centre of a set of concentric ripples on a pond was or was not a local minimum by choosing different points on successive ripples. I have tried hard to test that points between the selected points do indeed have intermediate function values.

4. The error surface

a) Previous work

The XOR problem is mentioned in virtually every text on neural nets. Literature on the error surface of our network has concentrated mainly on the practical matter of the circumstances under which back-propagation fails to find an acceptable solution. There have been widely different opinions. The experiments of Rumelhart *et al* (Rumelhart, McClelland, and the PDP Research Group 1986) suggested that there were (non-solution) local minima. Hirose *et al* (Hirose, Yamashita, and Huiya 1991) found that the range from which the weights were chosen (in our terms, the bound *b* mentioned above on page 12) had a profound effect. They found a u-shaped curve of non-convergence against *b*. This is reproduced here as Table 12.

They say that there is a local minimum around zero. "When the initial weights are very small, the calculations become trapped in this local minimum" (Hirose, Yamashita, and Huiya 1991, page 63). They do not give an explicit reason for non-convergence with weights greater than one, but imply that there are local minima there too. For example, they say (page 65) "How well this algorithm [the subject of their article] escapes local minima can be judged by … the result when the initial weights were chosen from the range (-5,5)."

Blum (Blum 1989) states that there is a manifold of local minima when the input weights to the output unit are the same, but Sprinkhuizen-Kuyper and Boers present a proof that these are saddlepoints and not minima (Sprinkhuizen-Kuyper and Boers 1994; Sprinkhuizen-Kuyper and Boers 1996).

Lisboa and Perantonis (Lisboa and Perantonis 1991), in what is claimed as a complete analytical solution of the problem, claim to have found "true local minima" of four different analytical types, and present five examples of such points (Lisboa and Perantonis 1991, Table 1). However, Sprinkhuizen-Kuyper and Boers give a proof that one of these is a saddlepoint (Sprinkhuizen-Kuyper and Boers 1996, page 1319). They also present a proof that all cases with finite weights and one input weight to the output unit (that is, weights 7 or 8 in our Table 6) equal to zero are saddlepoints.

Many previous discussions have used empirical investigations, but often these have

Range of weights, ±	Percentage non-convergence
0.05	100
0.25	10
0.5	0
1	0
1.5	20
2.5	30
5	50

Table 12: Non-convergence according to Hirose *et al* (adapted from their Figure 4)

been based on repeated runs of back-propagation using random starting points. Hirose *et al* is a typical example. Although much interesting information can be obtained in this way, there are two important shortcomings. Firstly, the method is not based on analysis of the surface and does not particularly help in understanding why the surface has a particular shape at a particular point. Secondly, the surface is so complicated that random explorations are very unlikely to exhaust the possible cases. As Figure 6 shows, the behaviour of our net using the step function is already fairly complex in three dimensions. As we shall see later, when the sigmoid is used each of the areas of that Figure ceases to be flat, ridges and trenches replace the lines of the Figure, and intersections between lines become peaks or holes. When all nine dimensions are considered the number of outcomes become very large. Also, some of these outcomes occupy quite small volumes in the space and thus are very likely to be missed by random probing.

A smaller number of investigations have approached the problem though a mathematical analysis of the surface. Lisboa and Perantonis take this route. In general, of course, analysis is preferable to brute force, but in this particular case they arrive at conclusions which we shall suggest are unhelpful. In addition, their concentration on the minima of the surface leads them to ignore the shape of the surface as a whole. In this paper we aim to combine an analysis of the whole surface with empirical investigations. In particular we shall look at the behaviour of back-propagation at various points on the surface.

b) Where the solutions are located

Our work with the step function, presented above, led us to the conclusion that the network computed XOR in 16 distinct volumes in the space. These are the volumes of the Boolean classes shown in Table 8. However, when we used the sigmoid we found that XOR solutions did not correspond to Boolean classes so neatly. So, are there still 16 volumes when the sigmoid is used?

Using a number of brute-force iterative procedures, many millions[2] of configurations were tested in a search for 'solutions': that is, for configurations which computed XOR with an SSE <0.01. These procedures explored the space of combinations of the nine weights, each weight taking on values in the range ± 30. Some of the procedures simply explored all points in regular nine-dimensional grids. Others explored areas around the boundaries of the Boolean classes, looking at points close to, and on either side of, the boundary lines shown in Figure 6.

These experiments yielded several thousand configurations that were solutions. One obvious hypothesis, arising from our previous work with the step function, would be that these solutions would be drawn from 16 basins in the surface of the SSE, one basin corresponding to each of the 16 Boolean classes of Table 8. We might expect, then, that application of back-propagation to these solutions would find the lowest point of each basin. This is not the case: a very high proportion of the solutions move only a short distance before stopping, and there is little sign (if any) of the solutions tending towards common minima. Even more disconcertingly, although all the Boolean classes of Table 8 are represented, solutions come from a much larger number of classes. However, further work revealed the following:

[2] Approximately 4×10^8

1. All the Boolean classes found in the solutions are either those listed in Table 8 or are derived from those of Table 8 by a change to only one of the hidden units (either the left or right hidden unit). For example, *b(4,d)* is one of the classes in the Table, and solutions are found that come from that class: so also are solutions from *b(0,d)* and *b(c,d),* where the hidden units from regions 0 and c respectively replace that from region 4.
2. The variant hidden unit always comes from a region that borders the original unit in the three-dimensional cube shown in Figure 6. Thus, in the previous example regions 0 and c are neighbours of region 4.
3. Single-linkage clustering of the classes of the solutions leads to a grouping of the solutions into 16 clusters, and the Euclidean centroids of these clusters correspond to the 16 Boolean classes of Table 8, with each centroid[3] lying in the middle of the Euclidean space of its Boolean class.
4. To test the homogeneity of the clusters, a 'walk' procedure was used. This procedure was applied to pairs of solutions and attempts to find a city-block walk (of at least two paces in each dimension) between the solutions such that every step of the walk is also a solution. In other words, the procedure looks for a (possibly curved) tunnel of solutions between its starting and end points, a tunnel that contains at least nine other solutions. It turned out that it is possible to 'walk' from every solution to the centroid of its cluster and to every other solution in its cluster, and it is not possible to 'walk' from any solution to the centroid of another cluster. Therefore, the clusters are distinct entities.

These findings show that our original hypothesis was very nearly correct. Using the sigmoid there are indeed exactly 16 solution-basins in the surface of the SSE, and these basins are indeed in the same regions of nine-dimensional space as the 16 Boolean classes that are solutions when the step function is used. However, our hypothesis was not completely correct. Firstly, the 'basins' are in fact 'trays': that is, they have flat floors (and hence solutions in them do not converge to a common point when iterative back-propagation is used). Secondly, the edges of these trays are not exactly where we would expect them to be: sometimes an edge includes a volume that we should expect not to provide solutions, and sometimes an edge excludes a volume that we should expect to provide solutions.

We can explore this last phenomenon graphically. Figure 7 shows the SSE surface in the upper-left quadrant of the plane of the input weights of the left-hand hidden unit of the solution at the centroid of *b(4,d)*. The weights for this solution are shown in (C5): the entries *x* and *y* are the dimensions in which the SSE is plotted in Figure 7. (At the centroid, those dimensions have the values *(-14,14).*)

x	***y***	7	-14	14	-7	14	-14	-7	(C5)

In Figure 7, we are looking at values of the SSE in an 80-by-80 grid of *x* and *y* positions, *x* ranging from –20 to 0 and *y* from 0 to 20. Each point in the grid has a corresponding letter, whose ASCII value is given by the formula

$$l = (\text{char})(\text{'a'-}(\text{int})(0.5 + 4\log_{10} s)) \qquad (14)$$

where *s* the SSE at the point. Thus, the letter 'a' represents a point where the SSE is 1.0, and higher letters correspond to lower SSEs, with 'h' and above denoting XOR solutions. Figure 7 should be compared with the second drawing in the first column

[3] The values of the weights are shown later, in Table 21.

of Figure 6: The plane plotted in Figure 7 corresponds to the top-left quadrant of that drawing, as can be seen from the correspondence between the annotating letters and lines of the two drawings.

Figure 7 shows the following:

1. The ‘basin’ of solutions using the sigmoid fits exactly within the region (the triangular area) that we would predict from our earlier analysis of the step function.
2. The rim of the basin, a transitional space where no-functions are computed, is *within* the region and thus reduces the size of the space where XOR is computed. This explains why we can have configurations that do not compute XOR when the sigmoid is used, despite being within a Boolean class that computes XOR with the step function. (We shall see a further reason in a moment.)
3. The ‘basin’ is indeed a ‘tray’ with a flat floor. We can also see that the rim of the tray is a smooth and steep escarpment.

This Figure, although it provides useful information, does not cast any light on the process whereby solutions are found *outside* the spaces of the Boolean classes that compute XOR. We can look at this question by constructing similar Figures for two such solutions. These are Figure 8, which plots a solution in *b(0,d)* (C6), and Figure 9, which plots one in *b(c,d)* (C7). These are the examples that we discussed earlier.

x	***y***	10	-10	10	-3	30	-10	-3	(C6)
x	***y***	10	-15	10	-10	15	-15	-5	(C7)

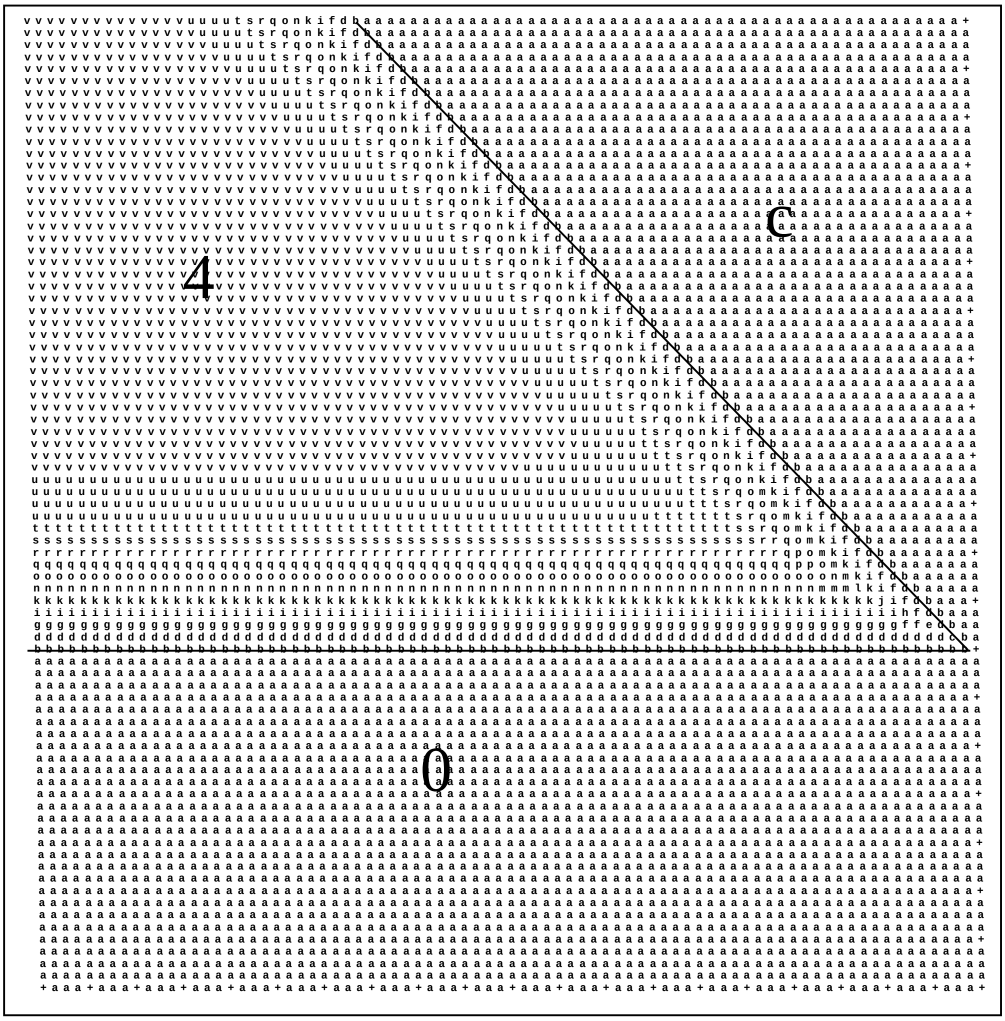

Figure 7: First two dimensions of a solution in *b(4,d)* (C5), upper-left quadrant

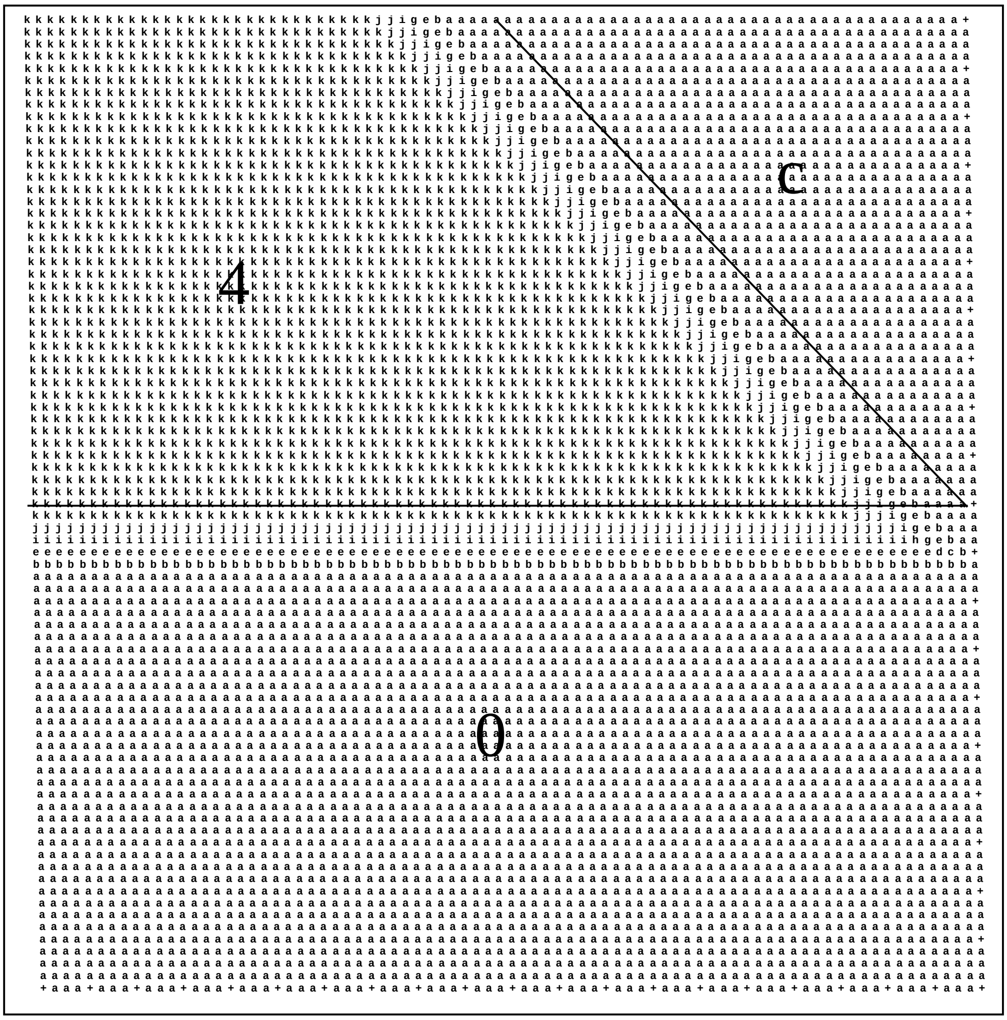

Figure 8: First two dimensions of a solution in *b(0,d)* (C6), upper-left quadrant

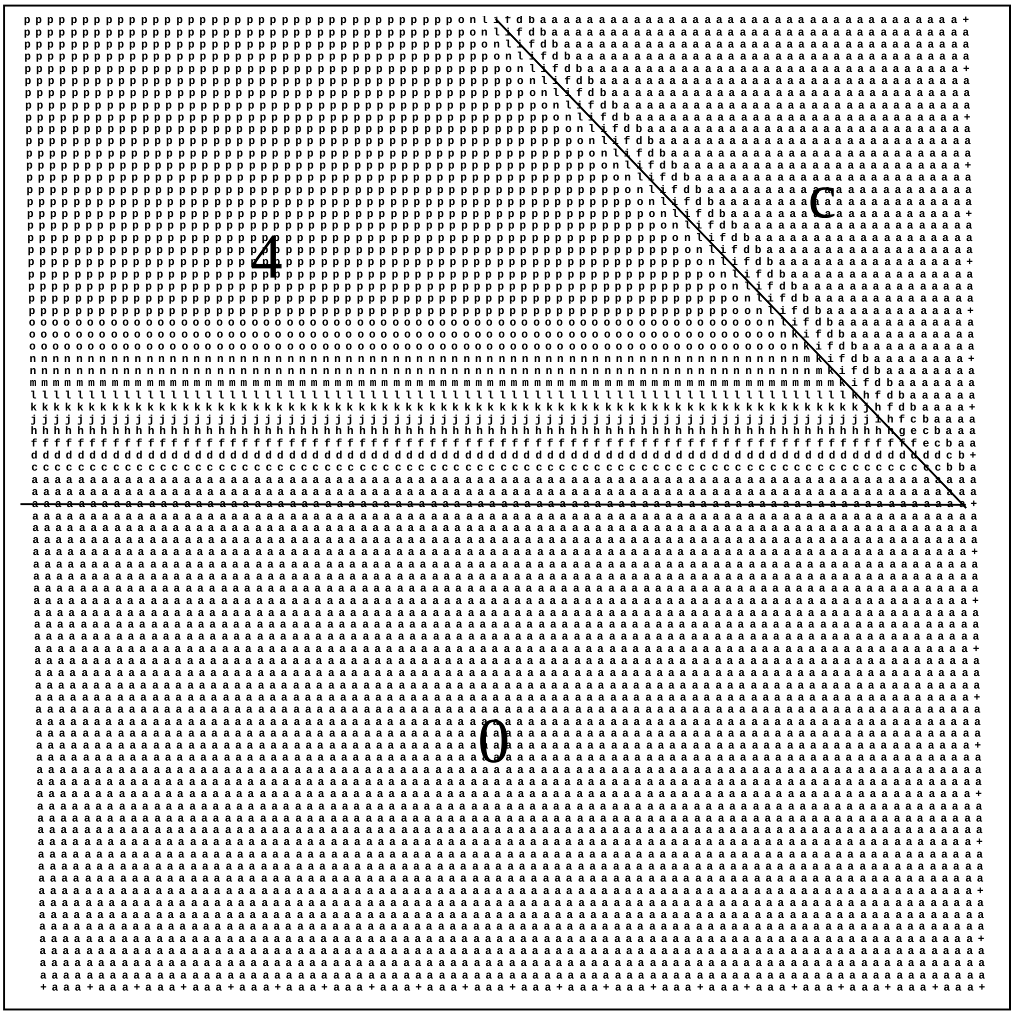

Figure 9: First two dimensions of a solution in *b(c,d)* (C7), upper-left quadrant

Figure 8 and Figure 9 show that the tray of solutions is displaced in the space of the left-hand hidden-unit (downwards in Figure 8 and upwards in Figure 9) when one of the other units in the net is not in the centre of its region. Thus in the case of Figure 8, in (C6) the output unit has unequal input weights; while for Figure 9 in (C7) the right-hand hidden unit has unequal weights. In both cases, the units are still in the same region as the corresponding units in (C5), but they are towards the edges of their regions. From the point of view of the performance of the net as a whole, this in turn displaces the tray of solutions that we are examining, so that

1. Some of the configurations that are solutions fall outside the expected Boolean class (and are in Boolean classes that normally compute functions other than XOR).
2. The Boolean class contains configurations that are not solutions.

Thus far in our exploration of the differences introduced by using the sigmoid (rather than the step function), we have concentrated on effects that have been attributable to the differences or ratios of the weights rather than to their magnitudes. We now turn to that question.

We have already noted that Figure 7 shows us that the trays of solutions have flat bases. Thus we can scale the values on the two dimensions plotted in the Figure (the input weights of the left-hand hidden unit) in a variety of ways without affecting the fit to XOR. However, it is clear from Figure 2 that the sigmoid will output values close to 0.5 for inputs close to zero, and so if the weights of a unit are all small then that unit will tend to produce an output of 0.5. In particular, if the output unit has small weights then the network as a whole will tend to compute 0.5 for any pattern of inputs. We can show this effect by taking (C5) and scaling it by dividing all the weights by four. The result is shown in Figure 10. In comparison with Figure 7, the tray of solutions moves within its quadrant and the escarpment becomes much less steep. The important change, however, is that none of the points in the diagram is an acceptable approximation to XOR: the value on the floor of the tray (designated by the letter ‘c’) has an SSE of about 0.2, above the value that we would use as the maximum for ‘a solution’.

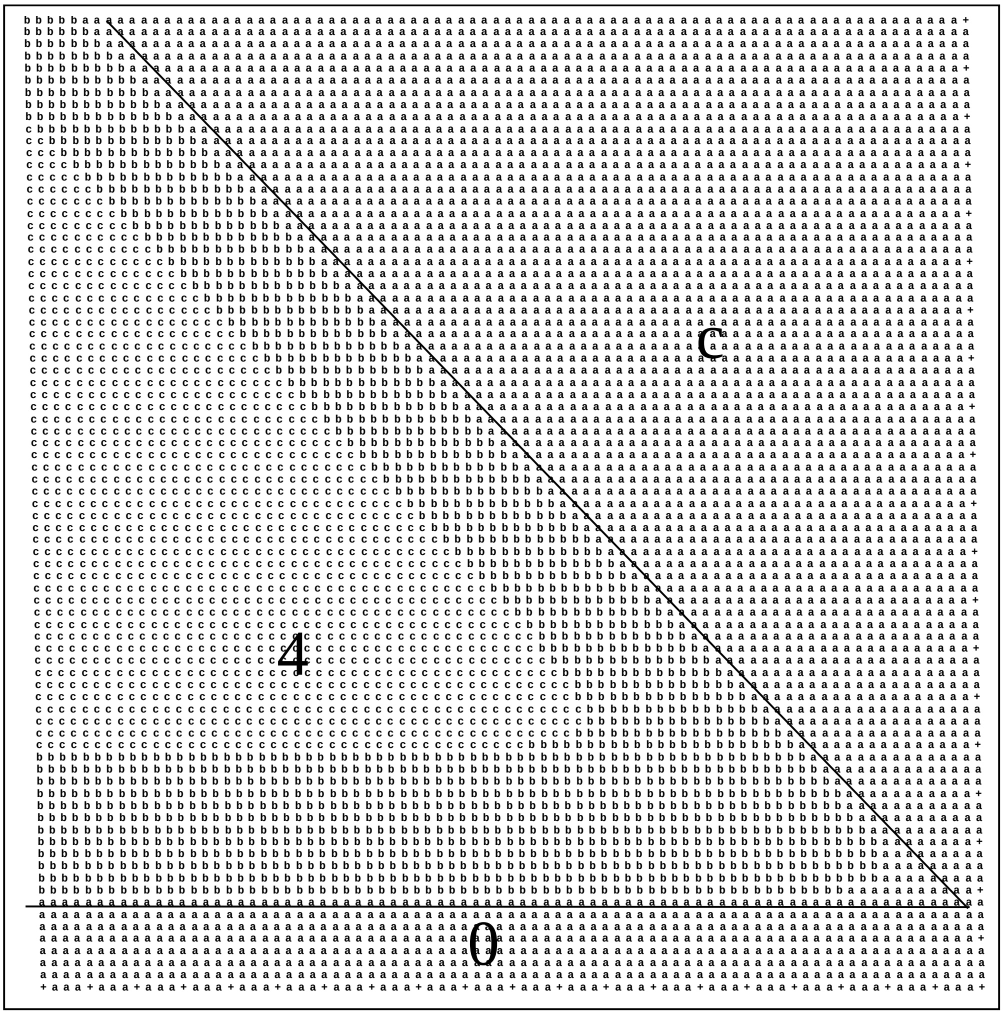

Figure 10: Figure 7 with all weights divided by four

c) Points near an escarpment

Consider the points ‘X’, ‘Y’ and ‘Z’ in Figure 11. These points are derived by changing a single dimension of a ‘standard’ solution (C5) in the centre of the tray of solutions originally shown in Figure 7. The values of the constant eight dimensions are shown in (C8), where the variable dimension is indicated by the letter ‘***y***’.

$$-14 \quad \boldsymbol{y} \quad 7 \quad -14 \quad 14 \quad -7 \quad 14 \quad -14 \quad -7 \qquad \text{(C8)}$$

Starting the back-propagation algorithm at the point ‘X’ $(y = 7)$, a solution is found in only three iterations. The rate of change is high, with the solution moving at an average crow-fly (Pythagorean) distance of 0.33 per iteration. The direction of movement takes place in five of the nine dimensions, of which the Y-axis of Figure 11 is one (as we would expect): in order of magnitude these changes are:

1. The point corresponding to the trial configuration moves up (north) on Figure 11 so that the point begins to descend the escarpment.
2. The threshold of the left-hand unit diminishes (this enlarges the area into which the configuration is moving: the edge of the escarpment shifts downward (south) to lie to the south of the point ‘X’.

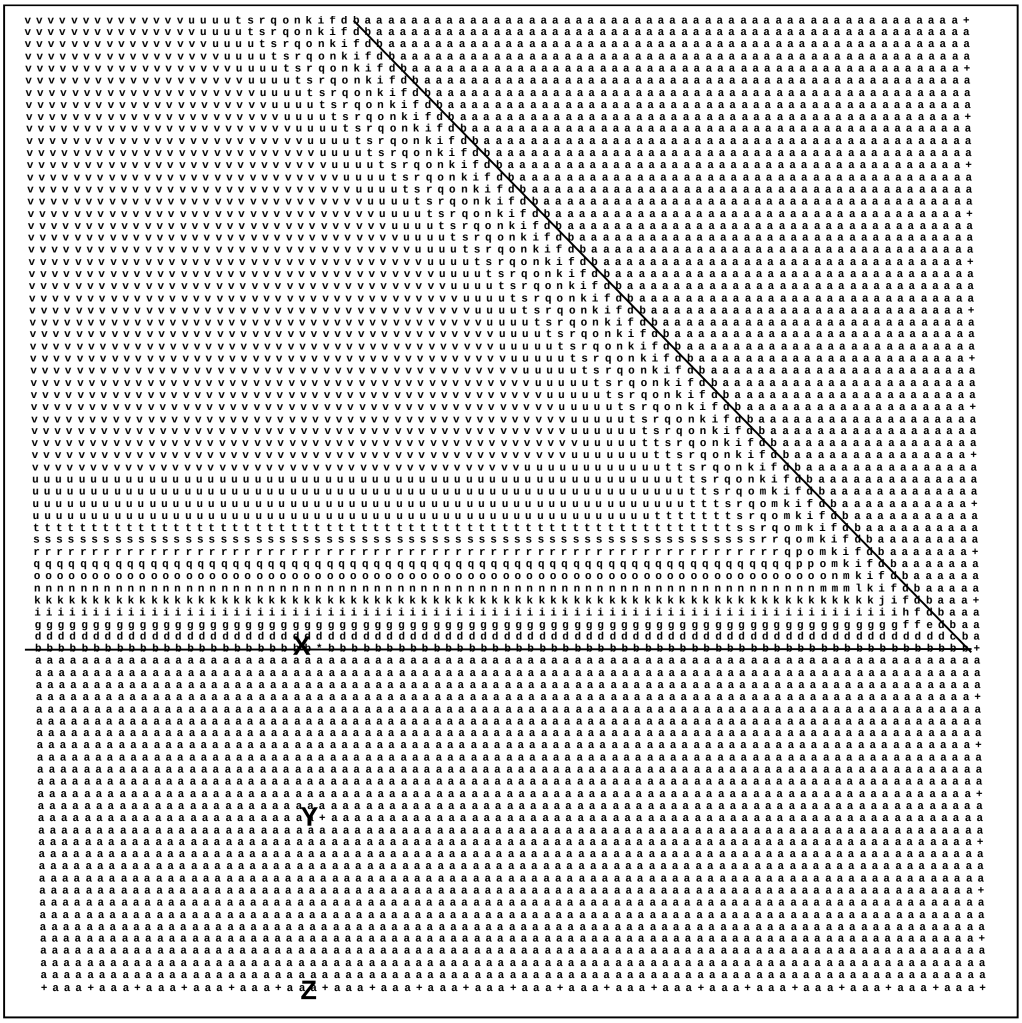

Figure 11: Starting-points for back-propagation minimisation

Value of *y*	Iterations	Mean distance per iteration
7.0	3	3.3
3.5	74	0.06
0.0	213	0.03
-1.0	394	0.01
-2.0	2182	0.004

Table 13: Effects of modifying starting point (alpha=0.95, eta=0.25)

3. All three weights of the output unit increase in magnitude and the unit becomes unsymmetrical, moving the triangular solution region towards the test configuration, in the manner that we have already seen from Figure 8: again this contributes to a southward shift of the escarpment's edge.

What happens if the point starts further away from the escarpment? From the point 'Y' on Figure 11 $(y = 3.5)$ we find that the configuration moves in just the same way as from the point 'X', but much more slowly: a solution is reached in 74 iterations at an average crow-fly (Pythagorean) distance of 0.06 per iteration. For most of this time the configuration is moving very slowly indeed across an almost flat plain until the northbound point meets the southbound escarpment edge, when speed picks up very rapidly.

Moving still further, from the point 'Z' on Figure 11 $(y = 0)$ we find that the behaviour is the same, but this time 213 iterations are needed at an average crow-fly distance of 0.03 per iteration. Carrying out a sequence of such experiments, successively moving the starting point further south, we obtain the results shown in Table 13.

It might be expected that the path followed by the algorithm would be a straight(ish) line. This is not the case, however, as Table 14 shows. This table shows the changes in the nine values of the configuration referred to in the last line of Table 13, where the starting configuration is to the south of the point 'Z' of Figure 11. These values are shown at 200-iteration intervals. In some of the columns, the direction of change reverses as the solution proceeds. This can be seen in the columns relating to the last value (the threshold weight) of the left-hand hidden unit and the last two values of the output unit. Indeed, the effect of the changes in these last two columns, with the values for the right-hand weight and the threshold moving together and then apart

Iter	LH Hidden Unit			RH Hidden Unit			Output Unit		
0	-14.00	-2.00	7.00	-14.00	14.00	-7.00	14.00	-14.00	-7.00
200	-14.00	-1.92	7.19	-13.83	14.17	-7.37	13.99	-10.84	-10.13
400	-14.00	-1.71	7.62	-13.62	14.38	-7.95	13.96	-10.81	-10.10
600	-14.00	-1.54	7.89	-13.52	14.48	-8.29	13.94	-10.79	-10.09
800	-14.00	-1.38	8.07	-13.45	14.55	-8.53	13.92	-10.78	-10.08
1000	-14.00	-1.22	8.21	-13.41	14.59	-8.72	13.91	-10.77	-10.07
1200	-14.00	-1.05	8.30	-13.37	14.63	-8.88	13.91	-10.76	-10.06
1600	-14.00	-0.86	8.36	-13.34	14.66	-9.01	13.90	-10.76	-10.06
1800	-14.00	-0.64	8.37	-13.32	14.68	-9.12	13.90	-10.75	-10.05
2000	-14.00	-0.34	8.32	-13.30	14.70	-9.23	13.91	-10.75	-10.05
2200	-14.00	0.13	8.12	-13.28	14.72	-9.32	13.92	-10.74	-10.04
2182	-14.00	4.85	3.97	-13.26	14.74	-9.40	14.23	-11.93	-8.85

Table 14: Progress of solution of last line of Table 13

again, is that the output unit moves very close to the point where it would compute f_f rather than f_b and then moves back again.

So far, we have managed to obtain a solution despite moving the starting point further and further from the tray of solutions. However, when the point is moved still further south, we run into difficulties. At $y = -3$ the algorithm, using the parameters shown in the caption to Table 13, apparently fails to find a solution in any practical number of iterations.

But is it stuck in a local minimum? The answer is “no.” In fact, the problem is caused by the fact that the gradient is now so small that the point is moving at a negligible speed. If we increase the parameter η (the learning rate) to (say) 4.0, then a solution can be obtained, albeit very slowly. Almost 20,000 iterations are required. As Table 15 shows, the path of the solution follows the same form as in the previous case: the output unit moves relatively quickly towards the boundary between f_b and f_f and hugs that boundary while the left-hand hidden unit crawls its way towards computing f_4 rather than the f_0 of the starting point. When the point finally reaches the escarpment then the output unit moves back towards its initial position, and the high value of η together with the steep slope of the escarpment carries the final solution much further in the final few iterations than in the previous 19,000.

Can we conclude that back-propagation will reach a solution from any point? No. If we change the configuration further in the direction which we have been pursuing, so that $y = -14$, say, giving

-14 -14 7 -14 14 -7 14 -14 -7 (C9)

then a solution cannot be reached with any set of parameters that I have tried so far. The configuration alters in most of the ways we saw in Table 15: the magnitudes of the thresholds of the hidden units increase, and its output unit executes the now-familiar move towards f_f, but then over many thousands of iterations the configuration drifts to a halt. If the momentum factor α is set to zero then it stops quite quickly, in the position given in (C10).

-14.00 -14.00 16.12 -16.58 17.47 -15.20 13.42 -11.71 -10.46 (C10)

Iter	LH Hidden Unit			RH Hidden Unit			Output Unit		
0	-14.00	-3.00	7.00	-14.00	14.00	-7.00	14.00	-14.00	-7.00
1000	-14.00	-2.33	10.80	-12.95	15.05	-10.97	13.73	-10.82	-9.82
2000	-14.00	-2.13	11.45	-12.90	15.13	-11.60	13.68	-10.80	-9.80
3000	-14.00	-1.99	11.82	-12.96	15.18	-11.90	13.65	-10.80	-9.80
4000	-14.00	-1.87	12.07	-13.10	15.22	-12.06	13.63	-10.82	-9.82
5000	-14.00	-1.76	12.26	-13.23	15.25	-12.18	13.62	-10.83	-9.84
6000	-14.00	-1.66	12.42	-13.35	15.29	-12.29	13.61	-10.85	-9.85
7000	-14.00	-1.56	12.54	-13.46	15.33	-12.39	13.60	-10.87	-9.87
8000	-14.00	-1.47	12.64	-13.56	15.36	-12.48	13.59	-10.88	-9.88
9000	-14.00	-1.37	12.72	-13.66	15.40	-12.56	13.59	-10.89	-9.90
10000	-14.00	-1.27	12.79	-13.74	15.43	-12.64	13.58	-10.91	-9.91
11000	-14.00	-1.17	12.85	-13.82	15.47	-12.71	13.58	-10.92	-9.92
12000	-14.00	-1.07	12.89	-13.89	15.50	-12.77	13.57	-10.93	-9.94
13000	-14.00	-0.95	12.92	-13.96	15.53	-12.83	13.57	-10.94	-9.95
14000	-14.00	-0.82	12.93	-14.03	15.56	-12.89	13.57	-10.96	-9.96
15000	-14.00	-0.68	12.92	-14.09	15.59	-12.95	13.57	-10.97	-9.97
16000	-14.00	-0.50	12.89	-14.15	15.62	-13.00	13.57	-10.98	-9.98
17000	-14.00	-0.28	12.82	-14.20	15.65	-13.05	13.58	-10.99	-9.99
18000	-14.00	0.05	12.65	-14.26	15.68	-13.10	13.59	-11.00	-10.00
19000	-14.00	0.80	12.14	-14.31	15.71	-13.14	13.63	-11.01	-10.01
19376	-14.00	8.42	4.77	-14.33	15.72	-13.16	14.17	-13.53	-7.50

Table 15: Progress of solution for *y = -3*

What we have found is a point at which the configuration is not computing XOR but is stationary (or, to be more exact, where the configuration moves in a closed path of four positions, one position after each of the four inputs of the XOR problem).

Common sense suggests that this point must be a "local minimum." This is not so, however. If we explore the space around the point by varying each value in (C10) by $\pm$ 0.5, generating 3^9 (19,683) starting positions, we find that although all are close to a stationary point and converge to it, there are 10,935 distinct stationary points, all with the same SSE. In all cases, the movement is in the last two dimensions: there is no movement in the first seven dimensions (that is, there is no gradient in any of those dimensions).

If we plot the SSE in these two dimensions, using for the other seven values those of the stationary point of (C10), thus:

-14.00 -14.00 16.12 -16.58 17.47 -15.20 13.42 ***x*** ***y*** (C11)

we obtain Figure 12. This Figure, unlike earlier ones, plots the whole plane in the square from (-20,-20) to (20,20) rather than just a quadrant. In a further change, the SSEs are indicated by lower-case letters chosen automatically by the mapping procedure so as to range from 'a' (highest value) to 't' (lowest value). Here 't' indicates an sserror of (about) 0.7 and 'a' one of 3.0 (none of the points is a solution to XOR, in other words). The position of (C10) is indicated on the plot by the letter 'A', and we can see that this point lies in a trench of the lowest values in the plane. This trench runs from the bottom left-hand corner of the plot towards the origin.

When the hypercube of starting configurations round the stationary (C10) move to the stationary points mentioned in the previous paragraph, it is to the floor of this trench that they move their ***x*** and ***y*** dimensions. For each of the 19,683 configurations in this experiment, the other seven dimensions remain the same: and, because all points along the floor of the trench are at the same depth, there are many possible finishing values of the ***x*** and ***y*** dimensions. These considerations, then, explain why we see as many stationary points as we do.

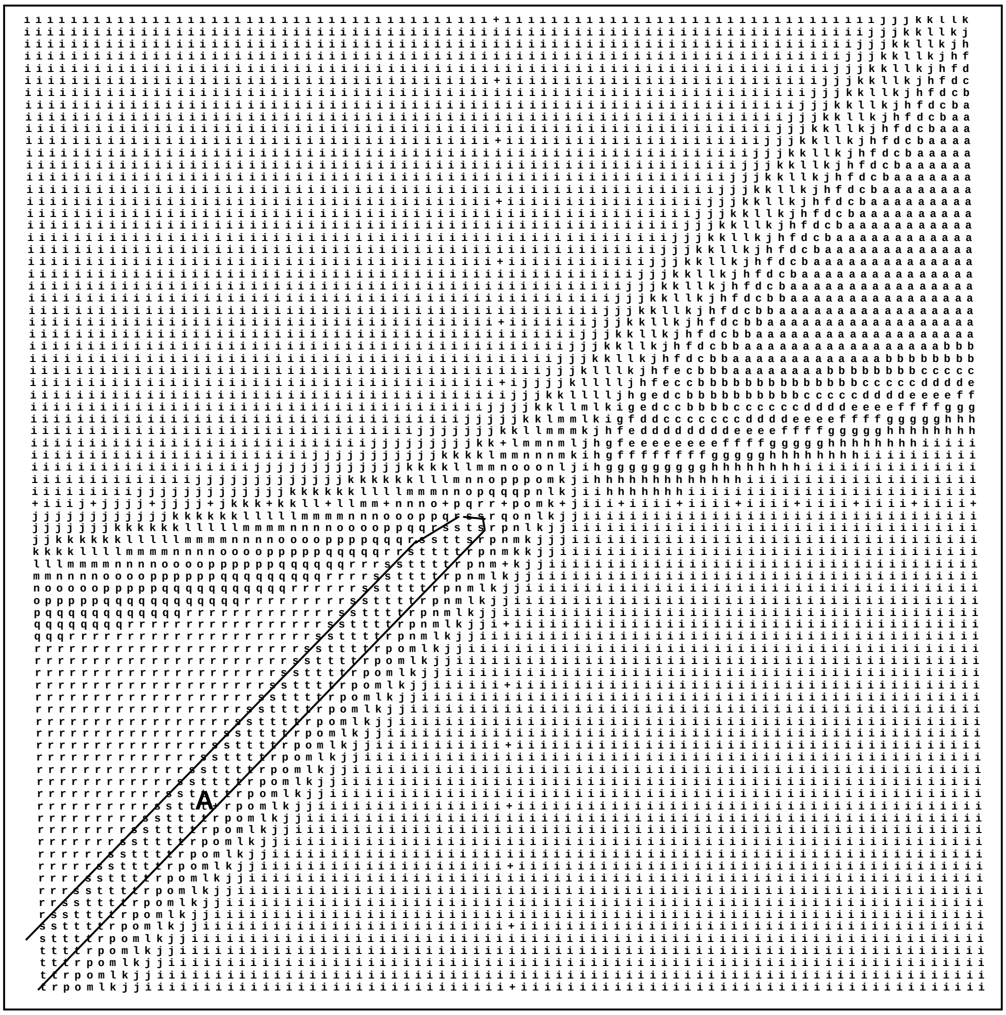

Figure 12: Trench of non-solution stationary points: plot of (C11).
Letter 'A' shows the position of (C10).

But is the floor of this trench a "local minimum"? No. It is a *global* minimum in terms of these two dimensions, holding the others constant, and a flat ledge in terms of the configuration as a whole. To demonstrate this, we can experiment by progressively moving the configuration to where we know a solution can be found: that is, by progressively increasing the dimension ***y*** of (C8) from –14 towards a positive value. Of course, we found the stationary configuration of (C10) from the standard solution in *b(4,d)* by progressively decreasing the ***y*** value, moving through the points 'X', 'Y' and 'Z' of Figure 11 and on to the configuration of (C9), from which back-propagation moved us to (C10), and so this new experiment should simply reverse the process. We expect to encounter the escarpment to the south of the solution region at some stage in these progressive increases in the ***y*** dimension.

This experiment was carried out as follows: the back-propagation algorithm was run on the configuration of

-14.00 ***y*** 16.12 -16.58 17.47 -15.20 13.42 -11.71 -10.46 (C15)

(which is the stationary point of (C10) with one variable dimension indicated by the letter '***y***') giving the ***y*** dimension the values –13, -12, -11 and so on. The results of this experiment were as follows: at each step, the new configuration was a new stationary point: the configuration simply settled like a drop of water placed on a flat sheet of glass. The flatness of the surface is strikingly shown by the fact that each new configuration had the same SSE as the original configuration, to (at least) six significant figures. This continued to apply until $y = 7$, when the point moved, slowly and then with rapidly accelerating steps, reaching a solution at

-14.00 11.45 11.67 -16.58 17.47 -15.20 13.78 -12.45 -9.72 (C16)

This solution is, of course, in the same region as all the others we have been discussing in the preceding paragraphs: a solution of Boolean class *b(4,d)*, that is, one in the area shown as '4' in Figure 7.

We see, then, that the process of finding a solution from the starting point of (C10) follows much the same path as that shown in Table 15: successively

1. The hidden units increase their thresholds, and the output unit changes its values to come close to f_f, reducing the SSE by moving the configuration into the trench shown in Figure 12. These are the changes in the move from (C9) to (C10), and make up what we might call a palliative reduction: it is the quickest way of reducing the SSE, but will have to be undone later in order to achieve a final solution.
2. The left-hand hidden unit changes the ***y*** value to move northward towards the area shown as '4' in Figure 7. This stage does what we may think of as the essential task of moving directly towards the solution region.
3. The left-hand hidden unit reduces its threshold, and the output unit changes back towards the middle of f_b to allow the floor of the area to be at a lower error level, thus undoing the work of step 1. The ***y*** value rapidly increases to move the point onto the floor of the tray of solutions. These are the changes in the move from (C15) to (C16).

The critical difference, however, is that from the starting point (C9) the algorithm is incapable of doing the work of the second step without human assistance: for a very long distance, the surface to be traversed is flat. Worse, the changes of the first step have *increased* the distance that must be traversed to reach an escarpment: from (C9) the ***y*** dimension needs to be increased by 11 units to (about) –3, but from (C10) it needs to be increased by 21 units to (about) 7. The human operator with knowledge of the XOR space can see which way the solution should move, but the algorithm cannot. The remedy suggested in all the basic texts is, of course, to use a momentum term: but this would be completely ineffective in this case. Firstly there is no gradient towards the escarpment and no momentum towards it can never develop; and, secondly, the only movement that takes place from points close to (C9) is movement that must be reversed later, and momentum would be counter-productive.

This sub-section has provided us with an example of the features that characterise the error surface: plateaux and trenches. In the next subsection we look at this more systematically.

d) Plateaux and trenches

The sixteen Boolean functions each have a "natural" SSE, given by the number of cells by which their truth table differs from the truth table of XOR. Thus, for example, the natural SSE of f_0, False, is 2 because its truth table is all zeros but the truth table of XOR has two ones. These natural SSEs are set out in Table 16, which also gives in its third column the number of Boolean classes computing each function and therefore the number of Boolean classes at the given levels.

Thus, using the step function the error surface would consist of 2,744 flat surfaces, one for each Boolean class, with each surface at the height shown in Table 16. Between surfaces, there would be discontinuous changes of level. When we use the sigmoid, however, the picture is more complicated:

1. At the origin (all weights zero) the SSE is 1. This is because the sigmoid produces 0.5 for summed inputs close to zero, and hence when all weights are zero all four cells of the output truth table approach 0.5, differing from the XOR truth table by ±0.5. Thus the SSE is $4 \times (0.5)^2$ i.e. 1.0.
2. The net is computing a continuous function and thus the error surface is smooth. It follows that the plateaux have smooth edges where a transition is made to the next plateau (or, towards the origin, to a height of 1).
3. The plateaux are not exactly flat. For example the eight surfaces where XOR is computed all have slight downward slopes as all weights are increased: as the weights increase the SSE drops. This is because the output of the sigmoid becomes closer to 0 or 1, and the output of the net approaches the ideal truth table.

To explore these phenomena, 14 sample sets of "ideal" weights were used, one for each of the functions computable by a single unit. They are "ideal" is the sense that they were chosen to represent points in the three-dimensional space of Figure 6 so that each point was as far away as possible from the others. The values are given in Table

Function	SSE	Number of Boolean classes
0	2	524
1	3	144
2	1	144
3	2	128
4	1	144
5	2	128
6	0	16
7	1	144
8	3	144
9	4	16
a	2	128
b	3	144
c	2	128
d	3	144
e	1	144
f	2	524
Total		*2744*

Table 16: Natural SSEs of the Boolean functions

Function	W1	W2	W3
0	-4	-4	2
1	-4	-4	-2
2	4	-4	2
3	0	-4	-2
4	-4	4	2
5	-4	0	-2
7	-4	-4	-6
8	4	4	6
a	4	0	2
b	4	-4	-2
c	0	4	2
d	-4	4	-2
e	4	4	2
f	4	4	-2

Table 17: "Ideal" single-unit weights for Boolean functions

Function	**Count**	**SSE**				
		Natural	Max	Mean	Min	Std Dev
0	524	2	2.000	1.964	1.929	.027
1	144	3	2.953	2.889	2.871	.019
2	144	1	1.001	.975	.965	.015
3	128	2	1.954	1.927	1.913	.013
4	144	1	1.001	.975	.965	.015
5	128	2	1.954	1.927	1.913	.013
6	16	0	.002	.002	.002	.000
7	144	1	1.001	.975	.960	.015
8	144	3	2.953	2.889	2.877	.019
9	16	4	3.824	3.824	3.824	.000
a	128	2	1.954	1.927	1.919	.013
b	144	3	2.953	2.889	2.871	.019
c	128	2	1.954	1.927	1.919	.013
d	144	3	2.953	2.889	2.871	.019
e	144	1	1.001	.975	.965	.015
f	524	2	2.000	1.963	1.918	.028

Table 18: Outputs of nets constructed from "ideal" units, by Boolean class

17. Each of the 2,744 Boolean classes could then be modelled by constructing a net using the appropriate units. Additionally, the effects of scaling all the weights in such a net could be explored.

We look first at the extent to which networks constructed in this way do actually approximate the “natural” SSEs. Table 18 shows the results when the weights are scaled by a factor of 2 (in order to saturate the sigmoid). Broadly, the 2,744 nets give the expected results. Grouping the classes by their expected output, we find that each group has on average a close approximation to its “natural” SSE, and the classes are tightly clustered within their groups. For example, the 524 classes computing f_0, False, have heights with a standard deviation of only 0.027. We note the following points:

1. classes whose natural SSE is 3 or 4 have computed SSEs less than the natural SSE
2. classes whose natural SSE is 2 have computed SSEs that are 2 or less
3. classes whose natural SSE is 1 have computed SSEs that straddle 1.

What happens as the weights are scaled (multiplied by a positive constant)? We know that if the weights of any configuration are scaled towards zeroes then the SSE approaches 1. Thus we would expect that scaling outwards from the origin we will see the SSE move from 1 and approach a final value asymptotically. This is indeed what happens.

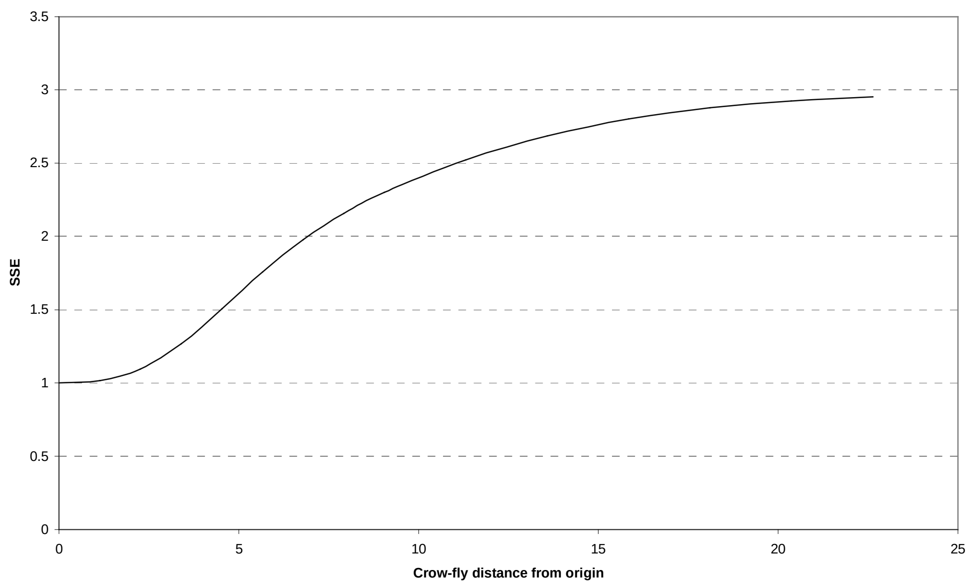

Figure 13: SSE of *e(d,d)*

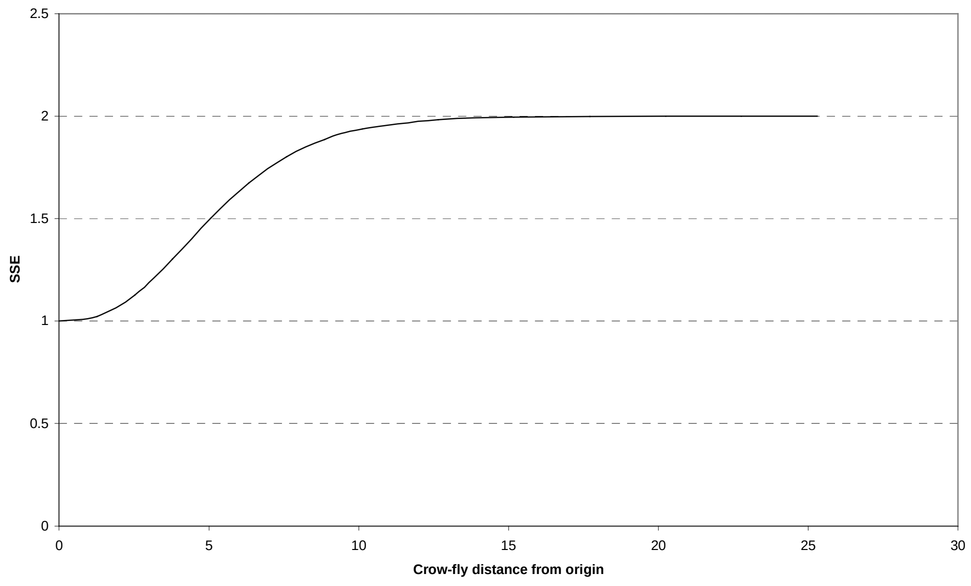

Figure 14: SSE of *7(0,0)*

Figure 13 shows what happens to the SSE as the weights of a class whose natural SSE is 3 (in this case *e(d,d,)*) are scaled towards the origin. We see that the SSE decreases smoothly towards 1.0 as the origin is approached. Compare this with Figure 14, showing a class whose natural SSE is 2 (in this case *f(0,0)*). We see that in the latter case the slope from the origin to the natural SSE is much steeper and hence the plateau is much larger and flatter, at least in the plane through the origin.

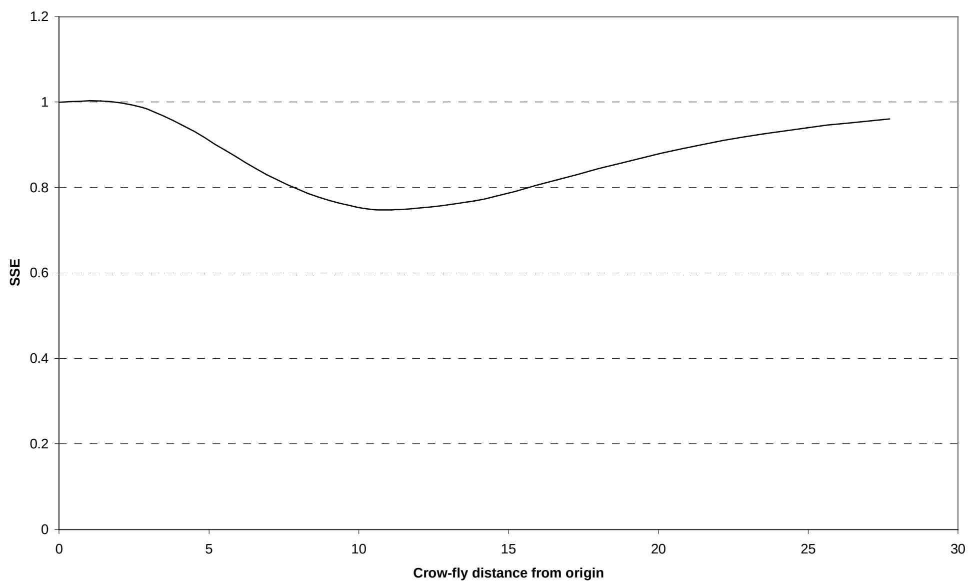

Figure 15: SSE of *8(7,f)*

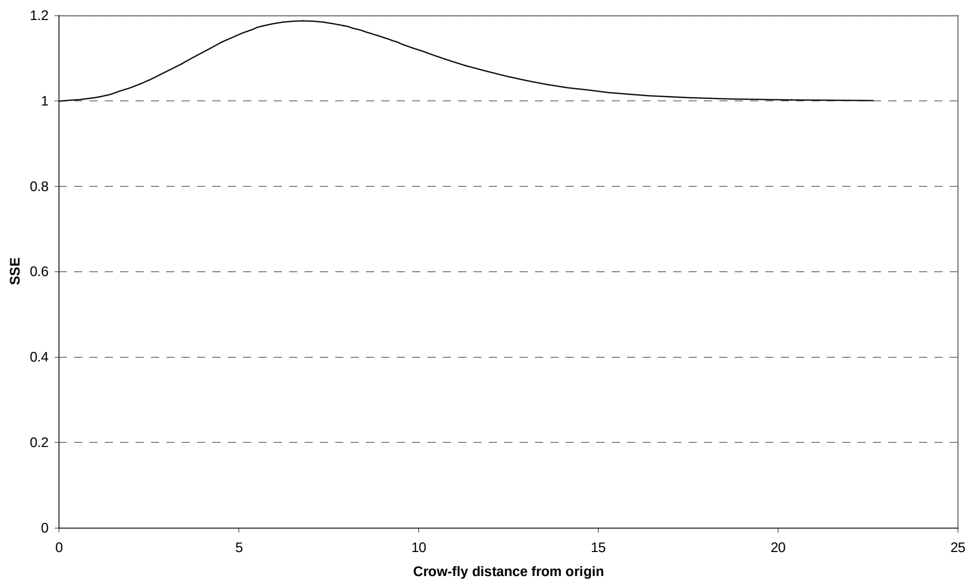

Figure 16: SSE of *7(c,a)*

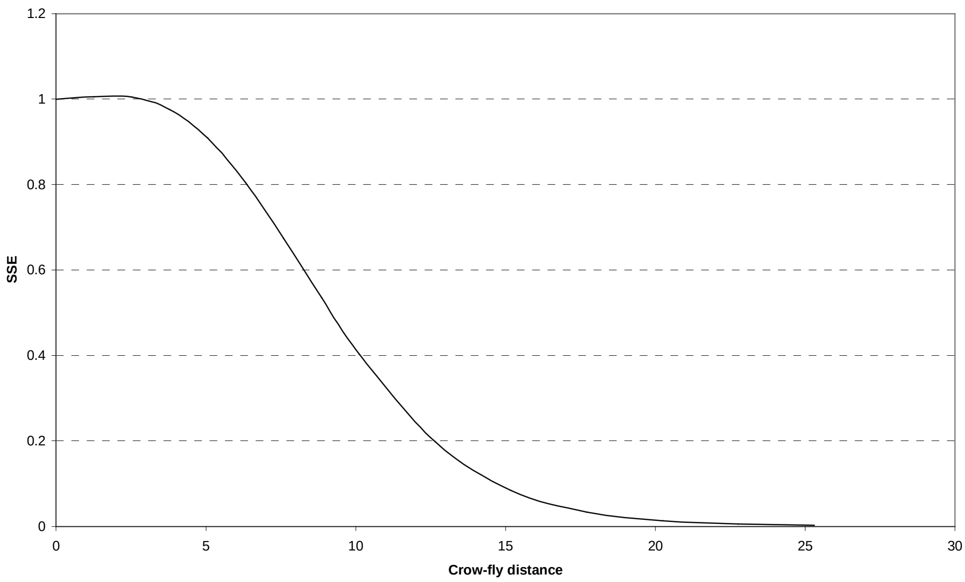

Figure 17: SSE of *2(7,1)*

One might imagine that Boolean classes whose natural SSE is 1 would have SSEs of 1 no matter how the weights were scaled. This is not the case, however, as we can see by scaling two examples, starting at the origin. Figure 15 shows that the SSE of *8(7,f)* falls to a minimum below 0.8 as it leaves the origin, before rising to its natural 1; while Figure 16 shows that *7(c,a)* rises to a maximum of almost 1.2 before falling to its natural 1. Similarly, although one might imagine that a configuration whose natural SSE is zero (i.e. an XOR solution) would show an SSE falling monotonically as the configuration is scaled away from the origin, we see from Figure 17 that *2(7,1)* rises very slightly before falling away towards zero. These effects are all caused by complex interactions between the weights, the truth tables of the functions and the behaviour of the sigmoid. The contributions of the different units of the net change at different rates if the weights are scaled down.

As configurations are scaled up (away from the origin) they asymptotically approach what we can call "saturated" values. Many (but not all) are close to their natural SSEs for crow-fly distances of 20 and above. This means that above this distance we find that the surface is characterised by large areas that are rather flat. Certainly at this distance there is only a very small gradient in the plane through the origin, but the gradient in other directions (orthogonal to that plane) will depend on the proximity to the area of another Boolean class. We can get some idea of the magnitude of these effects by rotating sample configurations round the origin. For example, if we move (C12) in a circle round the origin in the plane of its first and eighth dimensions, then we obtain Figure 18.

-8.0	8.0	-4.0	-8.0	8.0	-4.0	8.0	8.0	4.0	*e(d,d)*	(C12)

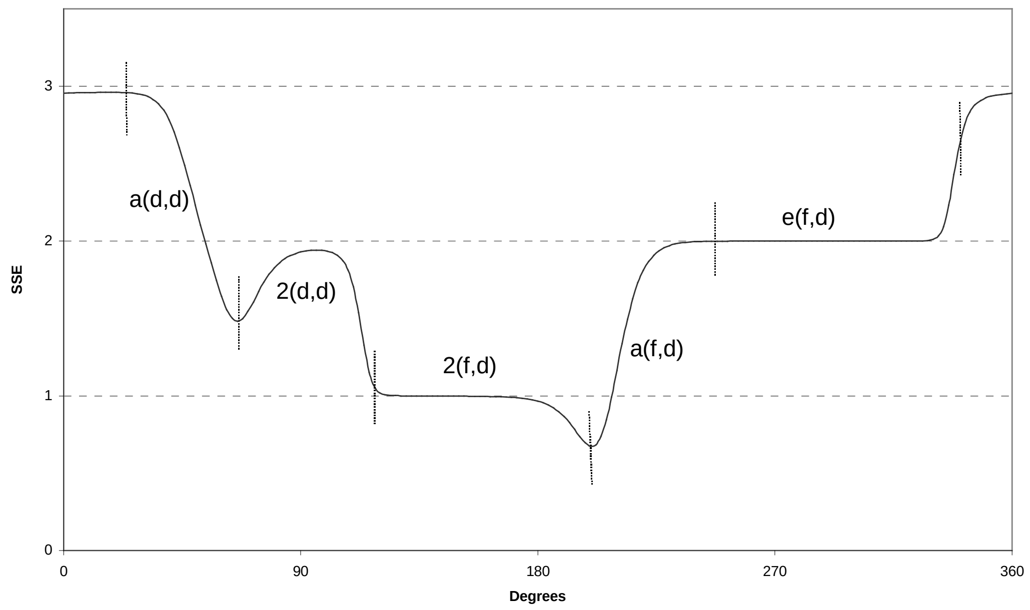

Figure 18: SSE as *e(d,d)* is rotated in the plane of its first and eighth dimensions

Because this transformation keeps constant the distance from the origin, it does not involve any of the scaling effects that we saw above: but there is much else to observe. Figure 18 shows that this rotation takes the configuration through a sequence of Boolean classes, through *a(d,d), 2(d,d), 2(f,d), a(f,d), e(f,d)* and back to *e(d,d)*. For some part of this journey the region does indeed appear to be flat, as in *e(f,d)*. But other parts are not at all flat: nearly all of the path through *a(d,d)* is on a steep slope. We can obtain some idea of the surface in nine-dimensional space if we superimpose eight circular paths round the origin, in different planes. We get these eight by looking at the planes of dimension 1 against each of the eight other dimensions (to get a complete idea we would need to examine all 36 possible rotational planes). These are shown in Figure 19.

Another way of viewing the surface at this point (C12) is to superimpose nine curves, each curve showing how the SSE changes if the configuration is moved ("disturbed") in a single dimension. This family of curves is shown in Figure 20.

These Figures suggest that (C12) is in fact at the top of a ridge, and that from (C12) (i.e. starting at either the extreme left or the extreme right of Figure 19, or outward from the central axis of Figure 20) there is a slope that leads down to a flat area at an SSE of 2. If asked to guess, we might suggest that back-propagation from (C12) would take that path and that it might then stick when it reaches the plateau. This is exactly what happens, and we arrive at[4] (C13).

-6.630	8.000	-5.370	-6.630	8.000	-5.370	8.185	8.185	0.385	*e(d,d)*	(C13)

[4] or close to it. Where exactly back-propagation will stop is of course a matter of the particular values used for the parameters α and η.

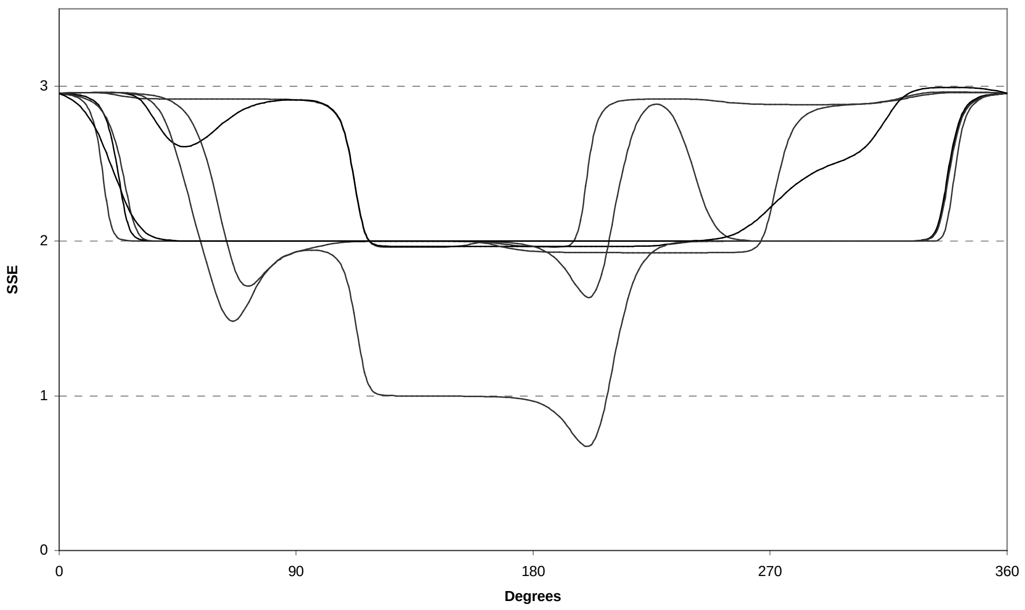

Figure 19: Family of SSE curves as *e(d,d)* is rotated in the plane of its first and second to ninth dimensions

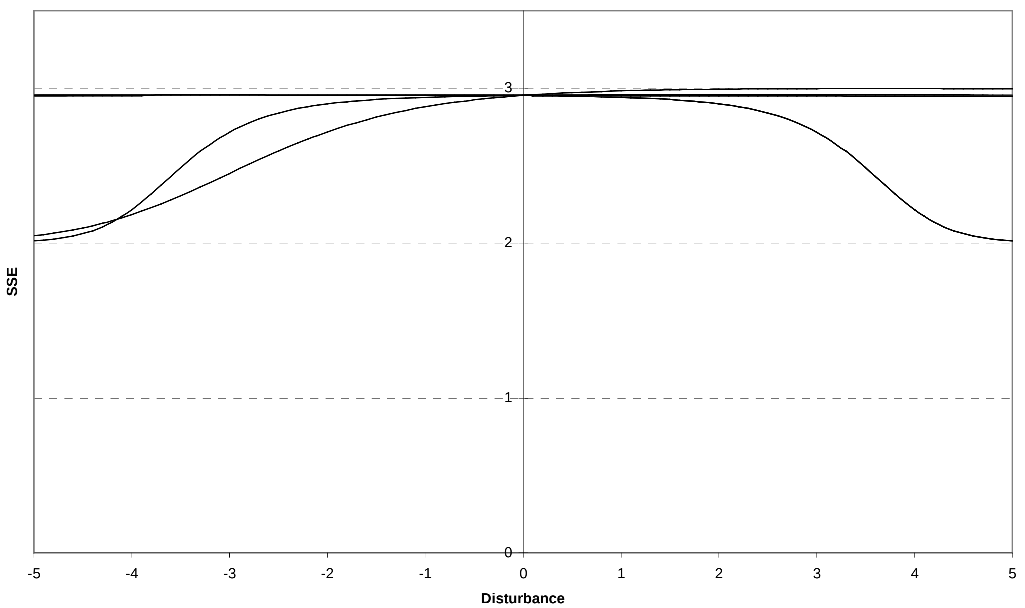

Figure 20: Family of SSE curves as *e(d,d)* is disturbed in each dimension

Re-drawing the family of disturbance curves from the new point, (C13), we obtain Figure 21. We see why back-propagation has stopped making progress: in all directions (as before, we follow curves outwards from the central axis of the Figure) the surface is flat, and in some directions it goes uphill. In one direction only it goes markedly upwards: this is the direction from which we came in our descent from (C12).

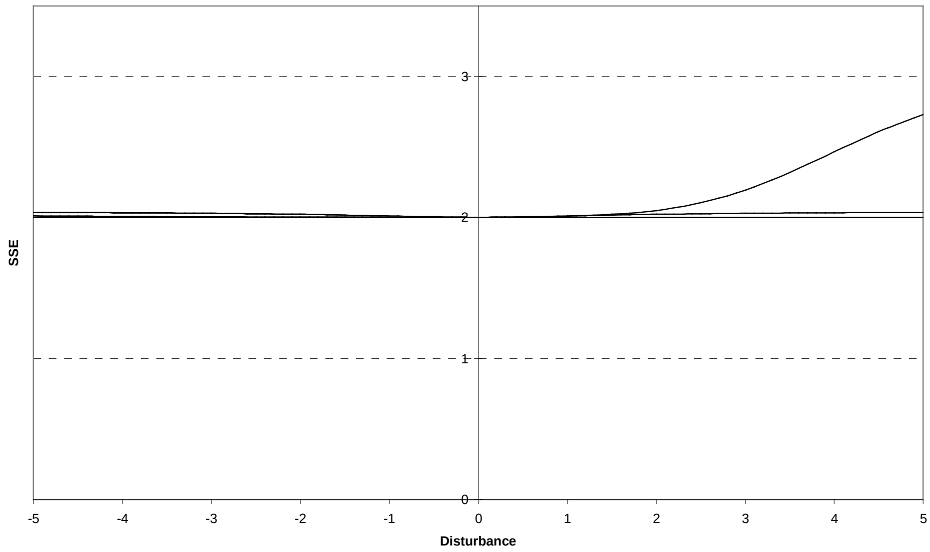

Figure 21: Family of SSE curves as (C13) is disturbed in each dimension

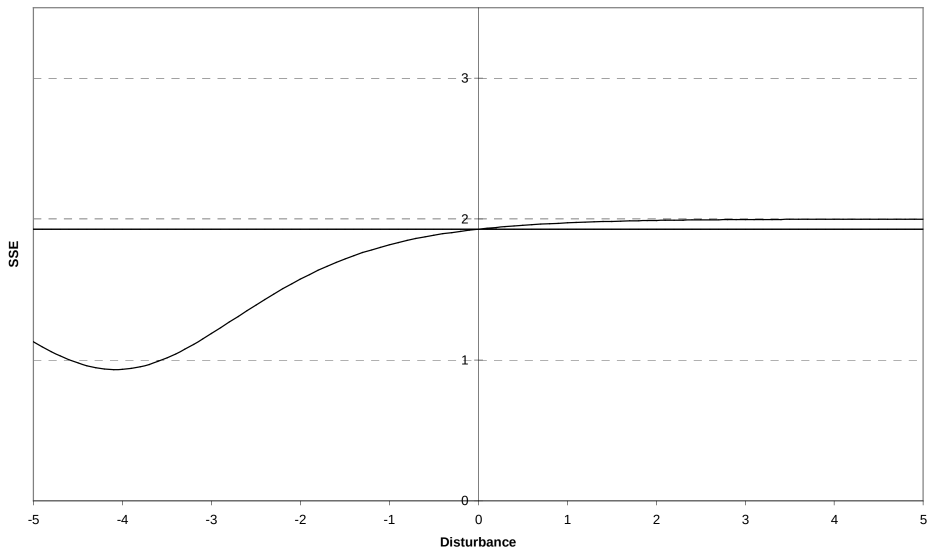

Figure 22: Family of SSE curves as (C15) is disturbed in each dimension

(C12) was an example of a configuration at the top of a ridge. We turn now to a striking example of a plateau: the configuration (C14).

-8.0 -8.0 4.0 -8.0 -8.0 4.0 -8.0 -8.0 -12.0 *7(0,0)* (C14)

The disturbance plot of this configuration (not shown here) appears to be a single flat line at a SSE of 2.0 from –5 to +5, representing nine superimposed lines all equally flat. In fact the lines are not quite flat - at the extremes of the plot two of them have dropped to 1.995 - but of course this cannot been seen on a plot scaled from 0.0 to 3.0. When back-propagation is started at this point, no movement takes place at all.

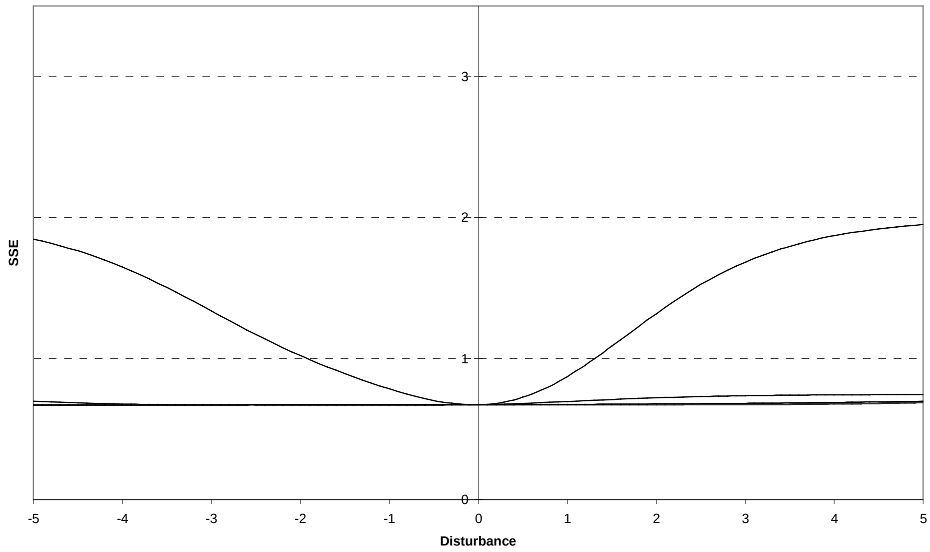

Figure 23: Family of SSE curves as (C16) is disturbed in each dimension

The two previous examples have been cases where back-propagation came to a halt on plateaux. Our next example, (C15), is a configuration that halts on the floor of a trench.

-8.0 -8.0 4.0 -8.0 -8.0 4.0 -8.0 -8.0 4.0 *0(0,0)* (C15)

Figure 22 shows the disturbance plots of this configuration, and the downward path of the configuration is clear on the left-hand side of the Figure.

Back-propagation on (C15) stops at (C16).

-8.0 -8.0 1.471 -8.0 -8.0 1.471 -8.34 -8.34 -.617 *1(0,0)* (C16)

The disturbance plots for (C16) are shown in Figure 23. The locally trench-like nature of the surface is shown by the fact that the dip in the SSE is in only one dimension, while the others are flat lines, some rising very slightly away from the point. Thus, the Figure looks as though this is a local minimum[5].

So far in this section we have looked at examples of configurations where back-propagation is unsuccessful in finding solutions in any reasonable number of iterations, and stops in non-solution regions of the space. Of course, there are configurations where back-propagation is successful, and there are configurations that are solutions already. An example of the first is (C17), which has a natural SSE of 3 but descends easily to an XOR solution.

-8.00 -8.00 4.000 -8.00 -8.00 -4.00 -8.00 8.000 4.000 *4(0,1)* (C17)

Figure 24 shows disturbance plots for (C17) and it is clear that descent is possible in any one of a number of dimensions (back-propagation will of course use the combination that gives the steepest descent), one of which will take us down to an SSE of 1. Moving our focus to that lowest point of Figure 24 (in fact this is (C17)

[5] In fact this is not a local minimum, and a solution can be reached from here (but not by back-propagation. We shall see later how this is done.

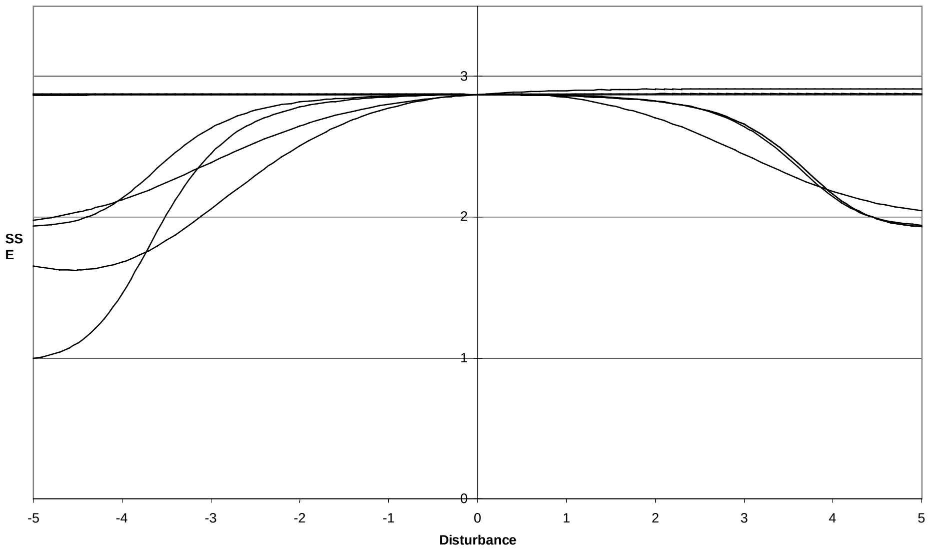

Figure 24: Family of SSE curves as (C17) is disturbed in each dimension

with the sixth dimension altered to –9) and drawing new disturbance plots we obtain Figure 25, which shows a dimension in which we can easily descend to a solution.

These examples show that back-propagation on the surface can, as is well known, give solutions: they also illustrate the two difficulties that can cause it to fail. These are firstly the existence of large flat areas with negligible gradients, and secondly the existence of strongly-marked trenches. We can examine the relative proportions of success and failure by trying back-propagation on each of the representatives of the 2744 Boolean classes constructed from the weights of Table 17. As before, we shall

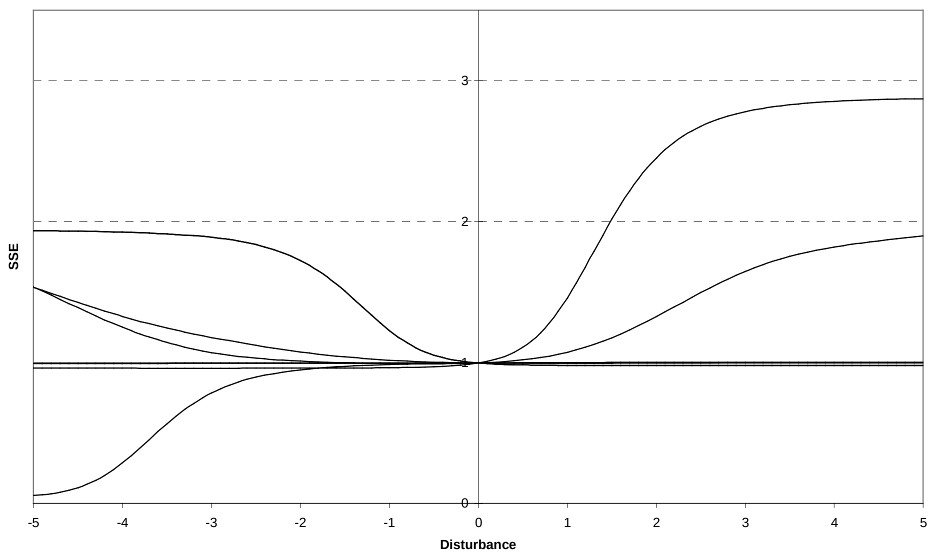

Figure 25: Family of SSE curves as the lowest point of Figure 24 is disturbed in each dimension

Original SSE	SSE after back-propagation (rounded)					All
	0.0	**0.5**	**1.0**	**1.5**	**2.0**	
0	16					16
1	144	288	144			576
2	224	491	567	8	270	1560
3	40	74	358		104	576
4					16	16
All	424	853	1069	8	390	2744

Table 19: Frequencies of outcomes of back-propagation on representatives of all Boolean Classes

scale these by 2.0 to increase saturation. This is the population of configurations from which the examples that we used above, (C12), (C14), (C15), and (C17), were drawn.

Table 19 shows the results of this experiment, rounding the SSE after back-propagation to the nearest 0.5, and Table 20 summarises the results.

The relative proportions of solutions and non-solutions shown in Table 20 are consistent with the results of other authors who have used random weights, such as (Hirose, Yamashita, and Huiya 1991). The figures are sensitive to scaling of the configurations, as we might expect: broadly, if the weights are scaled down then the number halted on plateaux is reduced both absolutely and as a proportion of non-solutions, while the proportion of non-solutions is itself reduced. Again, this is what Hirose *et al* found, although we shall in due course suggest that they are wrong about configurations close to the origin.

e) Connections to solutions

So far, we have used two techniques in our search for the solutions that can be reached from various starting-points. These have been back-propagation and human intuition guided by inspection of SSE plots such as Figure 7. In this sub-section, we describe the use of a new algorithmic approach. Given two configurations where the second has a lower SSE than the first, this approach uses a method that tries to *connect* the two configurations by finding a monotonically descending path between them. This algorithm is described in detail in Section 5, below, but we note the main points in the following paragraph.

The algorithm explores nine-dimensional city-block moves from the first configuration to the second using a step-size that is supplied as a parameter. In other words, it finds (or fails to find) a path consisting of discrete steps. Of course, in considering such a path we are principally concerned with the question as to whether

Outcome	Count	%
Already a solution	16	0.6
Converged on solution	408	14.9
Started on plateau, no movement	414	15.1
Halted after initial movement	1906	69.5
All	2744	100.1

Table 20: Summary of Table 19

there is a continuous path passing through the same points; or, if the algorithm fails to find a path, whether this means that a continuous path does not exist. Experience shows that, firstly, if the step-size is very coarse (1, for example) then the algorithm may find a path when in fact no continuous path exists, because the algorithm steps over an obstacle. However, the results that we shall report have been carefully tested using small step-sizes and we are confident that continuous paths exist in every case where a discrete path is reported. Secondly, under certain conditions the algorithm can fail to find paths in a reasonable number of iterations, even when a path exists. Typically, these are where a slope lies obliquely to the direction of travel.

Using this algorithm, we can see whether configurations can be moved down monotonically descending paths to one of the sixteen XOR solution regions. We call this a *connection* to a standard solution. Obviously, if a particular point can be connected to a solution then it cannot be (or be in the neighbourhood of) a local minimum.

The Boolean Classes of the sixteen solutions were listed above, in Table 8. The weights used for these solutions are shown in Table 21. These values are those of the centroids referred to above in Section 4, sub-section b), above. These configurations have very low SSEs, as the Table shows. Using them helps to avoid spurious non-connection with points on the edge of the region.

#	Weights									Class	SSE
1	-13	-13	-6	9	9	13	-14	-15	-8	*1(1,8)*	0.00000796
2	9	9	13	-13	-13	-6	-15	-14	-8	*1(8,1)*	0.00000796
3	-8	-8	-13	-13	-13	-6	15	-14	7	*2(7,1)*	0.00001027
4	-13	-13	-6	-8	-8	-13	-14	15	7	*4(1,7)*	0.00001027
5	13	13	6	9	9	13	14	-15	6	*2(e,8)*	0.00000796
6	9	9	13	13	13	6	-15	14	6	*4(8,e)*	0.00000796
7	14	-12	-6	-12	14	-6	-10	-10	-15	*7(b,d)*	0.00018891
8	-12	14	-6	14	-12	-6	-10	-10	-15	*7(d,b)*	0.00018891
9	-10	-10	-15	13	13	6	10	10	15	*8(7,e)*	0.00020241
10	13	13	6	-10	-10	-15	10	10	15	*8(e,7)*	0.00020241
11	11	-12	7	11	-11	-7	14	-13	-7	*b(2,b)*	0.00001425
12	11	-11	-7	11	-12	7	-13	14	-7	*d(b,2)*	0.00001425
13	-12	11	7	-11	12	-7	14	-13	-7	*b(4,d)*	0.00001416
14	-11	12	-7	-12	11	7	-13	14	-7	*d(d,4)*	0.00001416
15	11	-12	7	-12	11	7	14	14	6	*e(2,4)*	0.00001303
16	-12	11	7	11	-12	7	14	14	6	*e(4,2)*	0.00001303

Table 21: Weights for "standard" solutions

A crucial point is the following. The origin (the configuration with all values zero) connects to all 16 solutions. This startling result has been verified using very fine granularity (steps as small as 0.00001).

We can exploit this fact in dealing with initial configurations with SSE >=1. Instead of trying to connect them to the standard solutions, we can begin by trying to connect them to the origin: if they connect to the origin, then they can be connected to all 16 solutions. Our method attempts to do this by simple scaling of the weights ("mono-connecting" to the origin) or, if this fails, by using the full connection algorithm. Only if this fails does it revert to attempting connection to the individual standard solutions.

Table 22 shows the results of applying these methods to the 2744 test configurations. We note the following points:

Connection method	Original SSE (rounded)					All
	0	**1**	**2**	**3**	**4**	
Already a solution	16					16
Connectable via origin			1488	566	16	2070
Connectable to particular solutions(s)		576	72	10		658
All	16	576	1560	576	16	2744

Table 22: Connection (to any solution) of the 2744 trial configurations

- All can be connected to at least one solution.
- Of those with natural SSEs greater than 1, the great majority (96%) can be connected to the origin and therefore to all 16 standard solutions. I believe that all could be so connected, given perhaps an improved version of the connection algorithm, but I have not yet made a systematic attempt to prove it.
- Of those directly connected to solution(s), further work shows that the great majority (89%) can be connected to more than one standard solution (89% can be connected to at least two within 10,000 iterations using a step-size of 0.1). The average number of connections in this group is four. This figure could be increased by using more iterations, but it is not yet clear what the upper limit is. Even configurations with natural SSEs of 1 can typically be connected to more than one solution.

We have already seen (Table 20) what happens when back-propagation is used on the 2744 sample configurations: the majority halt on plateaux or in trenches. Clearly, it is of considerable interest to know whether these include any local minima: after all, if local minima exist, an excellent way of finding them would be to start back-propagation at representative points on the surface and examine the points where back-propagation stops unsuccessfully. With this in mind, it is very remarkable that the connectability of the 2744 points after back-propagation is essentially the same as the connectability before back-propagation. Table 23 gives the results of attempting to connect the 2744 back-propagation outcomes to standard solutions. We note the following points:

- All can be connected to at least one solution. That is, there are no local minima.
- Of those with SSEs greater than 1, the great majority can be connected to the origin and therefore to all 16 standard solutions. Again, it is conceivable that all could be so connected.
- Of those directly connected to solution(s), further work shows that many can be connected to more than one standard solution.

Connection method	SSE after back-propagation (rounded to nearest 0.5)					All
	0.0	0.5	1.0	1.5	2.0	
Already a solution	424					424
Connectable via origin			16	4	381	401
Connectable to particular solutions(s)		853	1053	4	9	1919
All	424	853	1069	8	390	2744

Table 23: Connection (to any solution) of the 2744 trial configurations after back-propagation

f) Claimed local minima

In the previous two sub-sections we took a set of configurations representing all possible combinations of three units, and found that

- For 85% of these configurations, back-propagation failed to find a solution in a practical number of iterations.
- However, none of these failures were in a local minimum. In all cases, the lowest point found by back-propagation could be connected to a solution by a monotonically-descending path. Indeed, many could be connected to several solutions.

In this section, we shall look at points that previous authors have claimed to be local minima.

We look first at the claim of Hirose *et al* (Hirose, Yamashita, and Huiya 1991) that the region around the origin is a local minimum. This is claim is based on taking random starting points within constrained intervals: see Table 12, above. I have not been able to reproduce this very strong effect of the magnitude of the weights. In relation to the area round the origin, it is certainly true that back-propagation starting at a high proportion of the volume round the origin will fail: but it is not true that this region contains local minima. As we have seen, the origin itself (the limiting point in this process of shrinking the weights) is connectable to all 16 standard solutions. Thus the volume round the origin contains up to 16 valleys leading outwards and down. Analytically, the surface round the origin, as everywhere else, is differentiable. My own experiments with very small weights have had results quite different from those of Hirose *et al*. An examination of (approximately) 20,000 configurations with small weights ($|w_i| \leq 0.025$) found no configurations that could not be connected to a solution.

Secondly, we look at the claimed minima of Lisboa and Perantonis (Lisboa and Perantonis 1991). Discussing the same network as us, they suggest that a number of (non-solution) local minima exist. Their discussion differs from ours in notation: they write the weights of a configuration in a different order and write the threshold weights with the opposite sign. A more important difference is that they use the convention that the network is attempting to reproduce the truth table of Table 24, in which conventional XOR table has each cell modified by small value δ, which is given some suitable value such as 0.1. This has a number of consequences, four of which we note now:

1. When their model is implemented in any practical computing device it must still include some other parameter defining an acceptably small deviation from a "correct" solution, so the introduction of δ complicates rather than simplifies matters.
2. We now have the odd situation that when a particular configuration of weights produces a truth table *closer* to the conventional table made up zeroes and ones, then computational work must be done to produce a (worse) result.
3. As we shall see shortly, we can find ourselves in a position where one or two of the training inputs are conveying no error information, although the current configuration is computing a no-function. This situation cannot arise using the

	1	$1-\delta$	δ
Y	0	δ	$1-\delta$
		0	1
		X	

Table 24: Alternative Truth Table for XOR

approach that we have taken so far, of measuring deviations from the "ideal" XOR truth table using a function that only approaches 0 or 1 as $x \to \pm\infty$.

4. What is in their terms a local minimum (a point surrounded by higher points) might be equivalent in our terms to a point in a trench whose floor approaches a non-solution minimum value as the values of weights approach infinity. For present purposes, this difference is not important: in both cases we should not be able to move from such a point to a solution without going uphill.

They present their local minima in terms of the four outputs produced by the network in response to the four possible input patterns, as follows: $(1,1) \to O^1$, $(1,0) \to O^2$, $(0,1) \to O^3$, $(0,0) \to O^4$. The cases in which they claim minima are produced are:

a) $O^1 = O^2 = O^3 = O^4 = \frac{1}{2}$

b) $O^1 = O^2 = \frac{1}{2}$, $O^4 = \delta$, $O^3 = 1-\delta$ and similar "solutions"[6] with $O^1 \leftrightarrow O^4$ and $O^2 \leftrightarrow O^3$

c) $O^1 = O^2 = O^3 = (2-\delta)/3$, $O^4 = \delta$ and the corresponding solution with $O^1 \leftrightarrow O^4$

d) $O^1 = O^3 = O^4 = (1+\delta)/3$, $O^2 = 1-\delta$ and the corresponding solution with $O^2 \leftrightarrow O^3$

In passing, we note that in case b) outputs O^4 and O^3 are exactly "correct" in terms of Table 24 and thus do not contribute to learning. In case c), the same thing applies to O^4 and in case d) to O^2.

Lisboa and Perantonis present five configurations as examples of these cases (Lisboa and Perantonis 1991, page 122). Although they do not say so explicitly, these examples are respectively of cases b), b), c), d), and a). We shall examine each of these examples in turn.

The first example, using our conventions and rounding to three decimal places, is the configuration

-5.521-13.690 -1.1419 -4.509 12.275 -4.736 -2.783 -5.057 -5.057 *3(1,f)* (C18)

This is an example of case b), and has a SSE of 0.52. Because our apparatus in this paper does not use the parameter δ, we should not expect this to be exactly a local minimum in our system: but if Lisboa and Perantonis are correct it should be very close to one, and this minimum would be reached by gradient descent. If back-propagation is used (with α=0, η=1) then this does look very like a local minimum. The configuration moves very slowly outwards but is at the bottom of a trench. After 10,000 iterations, it has reached (C19).

[6] A "solution" here means a solution to Lisboa and Perantonis' equations, and not a solution to the XOR problem.

-5.697-13.755 -2.328 -5.761 12.460 -3.749 -5.329 -5.695 -5.562 *3(1,d)* (C19)

Even using 100,000 iterations will not change this configuration significantly: it continues to drift outwards but the SSE remains above 0.5. However, this configuration (C19) can be connected[7] to *two* standard solutions: these are *1(1,8)* and *7(b,d),* rows 1 and 7 of Table 21. The fact that two solutions can be reached shows that this point is a saddle.

The second example is also of case b).

-11.521 -1.106-12.599 -11.900 4.010-11.707 4.593 -6.141 -1.549 *b(7,d)* (C20)

As with the first example, 10,000 iterations of back-propagation will move the configuration an insignificant distance and the SSE changes from 0.52 to just above 0.5. The result is (C21).

-11.724 -2.91 -12.799 -12.834 4.915-10.964 6.135 -7.44 -1.51 *b(7,d)* (C21)

However, (C21) can be connected to the standard solution *b(4,d)*, row 13 of Table 21. The point is therefore not a local minimum. There are indications that (C21) could also be connected to *2(7,1),* row 3, but the existing connection algorithm, using the parameters set out in Footnote 7 on page 49, does not terminate in a reasonable time and so this remains a conjecture.

The third example is of case c).

-13.709-13.709 -1.394 6.015 6.015 -5.217 -3.421 0.438 -0.109 *5(1,f)* (C22)

Again, back-propagation will not find a solution from here. Back-propagation will make the configuration drift to (C23), but not to a solution.

-14.178-14.238 -2.343 6.063 6.063 -5.305 -7.143 0.402 -0.231 *5(1,f)* (C23)

However, (C23) connects to *4(1,7), 4(8,e), 7(b,d), 7(d,b), 8(7,e).* These are rows 4, 6, 7, 8 and 9 of Table 21. The point is therefore a saddle.

The fourth example is of case d).

-10.425 8.558 10.280 -12.714 11.478 10.973 0.663 4.238 0.547 *e(0,4)* (C24)

Back-propagation will not find a solution from here. After 100,000 iterations we have

-10.518 9.069 10.105 -13.561 12.841 10.917 1.090 7.182 0.736 *e(0,4)* (C25)

However, (C25) connects to *4(1,7), b(4,d), d(d,4), 7(d,b), e(2,4).* These are rows 4, 13, 14 and 15 of Table 21. The point is therefore a saddle.

The fifth example is rather different. It of case a) and Sprinkhuizen-Kuyper and Boers (Sprinkhuizen-Kuyper and Boers 1996) show that case a) is a saddle rather than a local minimum. The example is:

0 0.483 -1.509 -0.572 0 0.896 0 0 0 *0(f,0)* (C26)

Unlike the earlier examples, this configuration has a SSE higher than 1.0 (albeit only very slightly). It connects to the origin and therefore to all sixteen standard solutions. This finding reinforces Sprinkhuizen-Kuyper and Boers' demonstration that the point is a saddle.

It can be seen from the previous discussion that none of the Lisboa & Perantonis "minima" are actually minima of the surface.

---

[7] With an effective step size of 0.001 (0.1 with two levels of inner iteration). These values are used throughout this subsection.

5. Conclusions

In this section we bring together material from previous parts of the paper to establish our conclusions about the XOR surface and the behaviour of the back-propagation algorithm when used on that surface.

We have seen that the back-propagation algorithm terminates unsuccessfully over a large part of the XOR surface. There are two typical cases.

In the first case, the point is in a region that is flat apart from a very gentle outward slope. This is characteristic of the central parts of non-solution Boolean classes. In these regions, locally the maximum SSE reduction is achieved by increasing the absolute value (the "saturation") of most or all of the weights. This is (of course) what the back-propagation algorithm does in such a situation. However, the strategy is unsuccessful because the SSE reduction is very small and becomes smaller with further iterations, tending towards zero. The algorithm is always unsuccessful in such a case, no matter what the values of α and η. Despite this, there is always a solution available via a descending path: this path does not run at right angles to the contours of the slope (the direction taken by the back-propagation algorithm) but initially nearly parallel to them. Such a path eventually comes to a point where it is directly above the solution to which it leads: from this point, back-propagation will succeed, usually quite rapidly.

In the second case, the point is in a trench that is the boundary between two non-solution Boolean classes. The trench is clearly visible in a plot of SSE in two dimensions of the configuration and runs diagonally across the plot. The floor of the trench is (almost) flat in the axis of the trench: the walls of the trench are steep. Such a trench will "capture" configurations that are being moved by back-propagation. If a momentum term is being used, a point that is captured in this way will spend many iterations rocking back and forward across the axis of the trench: these iterations will not contribute towards finding a solution. If a momentum term is not being used, then the principal movement of the point in a trench resembles the motion we saw in the preceding paragraph. There is a very gentle inflation of all or most of the weights which asymptotically reduces the SSE towards a non-solution value, and it appears that we are in a local minimum. Again, however, a solution is always available via a descending path. Typically, this begins with outward movement along the axis of the trench.

In both these cases, the path to a solution is normally curved. Sometimes the values on all dimensions move towards the solution values but at different (and changing) rates: but in many cases a dimension moves *away* from its final value and comes back towards it later. This dimension's initial movement away is needed to maintain SSE reduction as other dimensions change towards their solution values. Thus in some pairs of dimensions the movement may seem to be almost a closed loop. We see again that the use of a momentum term in back-propagation is inappropriate for this problem: because it promotes movement in a straight line, it may make it harder to follow a curved path.

Although these difficulties, and the analytical complexity of the surface, have led authors to claim that the surface has local minima, this is not true. There are exactly 16 minima and each is a solution.

6. Appendix: The monotonic connection algorithm

This appendix describes the algorithm to connect two configurations by a monotonic path.

1) There is a *from-configuration* and a *to-configuration*. We are attempting to reach the to-configuration in steps: at each step we change one or more of the nine values (the dimensions) of the from-configuration.
2) The algorithm *succeeds* if either
    a) The from-configuration is identical to the to-configuration (to within a small real tolerance, ε, in each dimension)
    b) The from-configuration has an SSE<0.3 and has the same Boolean Class as the to-configuration. This second criterion is an ad-hoc fix that recognises that a connection is imminent but avoids the possible waste of time involved in curving round a lower area of a solution basin lying between the from- and to-configurations.
3) The algorithm *fails* if either
    a) A user-selected maximum number of intermediate configurations have been considered
    b) The from-configuration has an SSE<0.3 but is remote from the to-configuration: specifically, if the Minkowski distance between the two is greater than 60. This is an ad-hoc fix that recognises that an SSE of 0.3 is only encountered in a solution basin, that the 16 solution basins are not connected, and that they are much smaller in diameter than 60. If this criterion is met, then the from-connection has been "captured" by the "wrong" configuration.
    c) There is no possible next step, in any direction, that meets the step criteria (defined in paragraph 9), below). (If, in other words, we are at a local minimum.)
4) We have a *step-size,* and any change to a dimension is by this amount (or, in the case of a forward move (defined in the next bullet) by `min(step-size, current-distance)`).
5) A *forward move* is either a change in a dimension of the from-configuration towards the to-configuration's value on that dimension, or, if the from and to-configurations already have the same value (to within ε) on that dimension, no change.
6) A *backward move* is a change in a dimension of the from-configuration away from the to-configuration's value on that dimension. If the from and to-configurations already have the same value on that dimension, this is called a *swerve* and can be either addition or subtraction of step-size.
7) At any proposed step, each dimension is marked for a forward move, a backward move, or no change.
8) The *number of forward moves (nforward)* at any proposed step is the number of forward moves minus the number of backward moves.
9) A proposed step (a *trial* configuration) *meets the step criteria* if `SSE(from)>=SSE(trial)>=SSE(to)` and the new configuration has not been visited before.

10) Here is the algorithm that finds the next step:

```
    For nforward:=9 downto –9
     Loop through all combinations of forward and backward moves
     that yield the current value of nforward
       Set up trial configuration as defined by the current
       combination of moves
         Repeat through swerves
           Step-counter++
           If step-counter>max-iter
             Exit (failure - too many steps considered)
           If trial configuration meets step criteria
             Exit (found next step)
           Until all swerves have been tried with both
           addition or subtraction of step-size
     End-loop
  End-for
  Exit (failure - local minimum)
```

11) The algorithm of the previous bullet is repeated until overall success (paragraph 1), above) or failure (paragraph 3), above)
12) An option is to impose the extra step criterion that a trial configuration must be connectable from the from-configuration by a recursive call at one-tenth of the step-size. (This is computationally cheaper than simply using the smaller step-size to begin with, as the recursive call can be placed last in the list of step criteria, resulting in fewer trials.)
13) The outermost loop is not entered if the from-configuration is already a solution in terms of paragraph 1), above.

An informal version of the algorithm is as follows: at each step a search is made for a new configuration that meets the criterion of monotonic descent towards the target. The first such configuration to be found is used. The search examines all possible moves (almost[8]), starting with moves that move directly towards the target configuration and ending with moves away from it. Thus, the algorithm will move directly towards the target if it can, but will manoeuvre sideways if necessary and can work its way round concave obstacles.

A number of observations can be made about this algorithm. The most important is the extent to which it returns false-positives and false-negatives. False-positives arise when the algorithm steps over a ridge, i.e. when intermediate points have SSEs that do not meet the criteria. My belief, based on considerable study of SSE plots, is that ridges do not occur in the XOR surface at very fine granularities. False positives are eliminated if a small enough step-size is used. I have used settings that I believe to be extremely conservative.

False negatives arise quite easily. If the contours of the surface are very oblique to the direction of travel, then it becomes computationally very expensive to find a path across the surface. Experiments on the XOR surface show that there are some paths that take very large numbers of iterations. When exploring whole families of such paths, runs times can rise to days. When the maximum number of iterations is set to values that allow reasonable run-times, then false negatives are created.

[8] Not all combinations of swerves are tested. Also, the user can select a limit on the number of iterations that will prevent some possibilities being examined.

7. Reference List

1. Blum, E. K. 1989. Approximation of boolean functions by sigmoidal networks: Part 1: XOR and other two-variable functions. *Neural Computation* 1: 532-40.

2. Bryson, A. E., and Y.-C. Ho. 1969. *Applied Optimal Control*. New York: Blaisdell.

3. Hertz, John, Anders Krogh, and Richard G. Palmer. 1991. *Introduction to the theory of neural computation*. Reading, Massachusetts: Addison-Wesley.

4. Hirose, Yoshio, Koichi Yamashita, and Shimpei Huiya. 1991. Back-propagation algorithm which varies the number of hidden units. *Neural Networks* 4: 61-66.

5. Lisboa, P. J. G., and S. J. Perantonis. 1991. Complete Solution Of The Local Minima In The XOR Problem. *Network - Computation In Neural Systems* 2, no. 1: 119-24.

6. McClelland, James L., and David E. Rumelhart. 1988. *Explorations in parallel distributed processing: a handbook of models, programs, and exercises*. Cambridge, Massachusetts: MIT Press.

7. Minsky, Marvin Lee, and Seymour Papert. 1969. *Perceptrons: an introduction to computational geometry*. 1st ed. Cambridge, Massachusetts: MIT Press.

8. Rosenblatt, Frank. 1960. *On the convergence of reinforcement procedures in simple perceptrons,* VG-1196-G-4. Cornell Aeronautical Laboratory, Ithaca, New York.

9. Rumelhart, David E., James L. McClelland, and the PDP Research Group. 1986. *Parallel Distributed Processing: explorations in the microstructure of cognition*. Cambridge, Massachusetts: MIT Press.

10. Sprinkhuizen-Kuyper, Ida G., and Egbert J. W. Boers. 1994. *A comment on a paper of Blum: Blum's "local minima" are Saddle Points*, Tech Rep 94-34. Department of Computer Science, Leiden University, Leiden, The Netherlands.

11. ———. 1996. The Error Surface Of The Simplest XOR Network Has Only Global Minima. *Neural Computation* 8, no. 6: 1301-20.