

An Introduction to Connectionist Computing

Chris Thornton

Cognitive and Computing Sciences
University of Sussex
Brighton
BN1 9QH
UK

Email: Christopher.Thornton@firenet.uk.com

Tel: (44)1273 678856

May 21, 2003

Abstract

The article provides an introduction to *connectionist computing*. It shows that connectionist methods differ from conventional methods mainly in the type of virtual machine which they exploit. Whereas conventional methods employ virtual machines founded on the von Neumann architecture, connectionist methods employ highly parallel machines made up of many simple processing elements connected together in a network. The article also presents illustrative examples of learning in connectionist networks are presented.

1 Introduction

There has recently been a resurgence of interest in connectionism as a computational method. This was partly triggered by new developments in automatic programming procedures for complex connectionist architectures. Such architectures are difficult to manipulate manually and so the development of novel automatic programming (ie. learning) methods has greatly increased the viability of connectionist computing in general.

Connectionist computing (CC) differs from conventional computing in a number of ways; but defining a clear boundary between the two types of computation is far from easy. For present purposes, we will assume that a *computational mechanism* is any system which systematically maps inputs onto outputs. And we will assume that *computation* is simply the behaviour of a computational mechanism. In this context we can broadly delineate the distinction between the two types of computation by differentiating between the main components involved.

In conventional computing, the components of interest are the symbols representing behaviours of the system (ie. instructions) and the symbols representing data objects. Computation in the conventional context involves the application of instructions to data objects.

In connectionist computing there are two main types of component to consider: firstly there are neuron-like *units*; secondly there are *links* which connect one unit to another. An assembly of such units and links is called a *neural network*. As devices the neuron-like units are extremely simple. Typically they store a certain level of *activation*, represented as a real number between 0 and 1. The links are capable of propagating activation from one unit to another. Computation in the connectionist context, then, involves activation being fed into and propagating through the network. To summarise, conventional computing involves instructions being applied to data objects; connectionist computing involves activation passing through a network of neuron-like units.

Although the differences between the two methods are quite striking, it is worth emphasising that connectionist computing is exactly like conventional computing in the sense that it is not committed to any particular implementation or to any particular type of hardware. It is a general method which can be embedded in a real, physical machine or any suitable virtual machine. In fact in a large proportion of current research, connectionist mechanisms are implemented as virtual machines running on top of conventional mechanisms (ie. they are implemented as programs running under a symbolic language such as C or Lisp).

The two methods are also equivalent in terms of theoretical power. As long ago as 1943 McCulloch and Pitts proved that networks of simple neuron-like units (modelled as threshold functions) were capable in principle of computing any desired boolean function [1]. That is to say, given a particular boolean function, it is always possible to construct a neural network which will compute that function.

Of course the phrase **compute that function** has a special meaning in this context. Computing a function using a neural network involves re-representing inputs and outputs as activation values. Presenting input involves setting the activation levels of certain *input units* (referred to as *clamping* an input vector onto the input units) while obtaining output entails reading the activation levels of certain *output units*. The nature of the function computed (ie. the output obtained) all depends on the way in which activation flowed through the network from input units to output units. That is to say, it all depends on the connections between units, hence the name **connectionist computing**.

2 Hand-coding v. learning

As we noted above, the process of configuring a neural network is the connectionist equivalent of programming. But, as such, it is programming at an extremely low-level. The closest equivalent in the conventional domain might be programming a von Neumann machine at the machine code or microcode

level. Nevertheless, for fairly simple functions hand-configuring the network is a viable proposition. Let us consider how we might configure a neural network which will function as a simple adder.

missing file cnctnst_cmptng_adder

We imagine that we have an infinite stock of neuron-like units and an infinite stock of links. Each unit has a threshold associated with it and maintains one of two levels of activation. It is either fully active (*on*) or completely inactive (*off*). Links are directed: they lead *from* one unit *to* another and are able to propagate activation in that direction only. The proportion of activation propagated is fixed by an *activation function* which in this example has a very simple linear form. It dictates that exactly half of a unit's activation is propagated down all links leading away from the unit in each time step.

Given these components we can configure a simple network adder for initial range of integers as follows. We have two sets of 5 input units and one set of 10 output units arranged as an ordered sequence. We set the links such that each output unit has a link running to it from each input unit. The arrangement is shown in Figure ?. Here and in all other figures representing networks, the circles represent the units and the lines represent connections between them. Circles in the lower part of the figure represent input units. Circles in the upper part of the figure represent output units. Thus activation flows upwards in these diagrams.

Adding two numbers n and m (where $n < 5$ and $m < 5$) using this network will involve activating n units in the first set of input units and m units in the second set of input units and then finding the highest (rightmost) active output unit. To get sensible addition behaviour we should set the thresholds in the outputs units so that the k 'th output unit only comes on if the total number of input units turned on is greater than or equal to k . Performing addition with the final network is simply a matter of turning on the appropriate input units and then finding the highest activated output unit.

Of course this is a trivial example which does not in any sense show the power of CC as a method. To do this we need to look at more sophisticated techniques.

3 Hopfield nets

Hopfield has described a CC method [2] that is applicable to any constraint optimisation problem which can be viewed as the attempt to find a set of mutually-supportive hypotheses which are compatible with some given evidence. Hopfield's technique, which is in fact a type of relaxation process, uses an analogy in which the set of hypotheses are likened to a set of molecules in a simple physical system, eg. a gas. In such a system, there are certain constraints affecting the interactions that molecules may have with one another. The energy of the system is effectively the degree to which the constraints are violated overall. Over time the gas gradually settles down into a state in which the behaviour of the

molecules is maximally compatible; ie. a state in which the constraints are least violated.

Hopfield showed how this type of relaxation process can be implemented in a neural network. In the implementation each unit represents a single hypothesis and therefore corresponds to a single molecule of the physical system. The weights on the links represent the constraints between the hypotheses. Each link is *symmetric*; that is to say activation may flow along it in either direction. Units can either be *on* (activation = 1) or *off* (activation = 0). In the simplest case a Hopfield net is completely interconnected. Every unit is linked to every other unit; see Figure ?.

missing file `cnctnst_cmptng_hopfield_net`

Imagine that we have a set of hypotheses H and a set of **evidential** constraints. Each constraint is a number which specifies the degree to which one hypothesis supports another. There is one constraint for every pair in $H \times H$. We also have some known facts F which take the form of certain hypotheses. The goal is to find a maximally compatible configuration of hypotheses in H given the constraint that all of F are true.

Hopfield's technique involves a fairly straightforward form of hand-coding. We configure the network such that there is one unit for each hypothesis and that there is a link from every unit to every other unit (ie. such that the network is completely interconnected). We set the weights such that size of a weight between two units is an indication of the degree to which the hypothesis corresponding to one supports the hypothesis corresponding to the other. We arrange things such that, in each time step, each unit either turns completely *on* (activation = 1) or completely *off* (activation = 0) depending on whether the net activation it is receiving is above some threshold.

To make the network perform constraint optimisation we **clamp** those units which correspond to the hypotheses known to be true (members of F) into the *on* state and let the network run. If units fire (ie. measure their net input and then turn on or off) in such a way that each unit has an equal probability of firing in any given time step, the network is guaranteed to gradually settle into a local incompatibility minimum [2]; ie. a state in which the active units are broadly compatible. This conclusion may be more or less obvious. However, the novelty of Hopfield's contribution was that it derived the conclusion on formal, mathematical grounds.

4 Measuring global compatibility

Consider a pair of units s_i and s_j . The presence of a positive weight on the link between them indicates that the hypothesis represented by one unit *supports* the hypothesis represented by the other unit, and vice versa. In effect, it indicates that the states of the two units should have the same signs. The level of compatibility is increased if the two states do have the same signs.

Conversely, the presence of a negative weight on the link indicates that one hypothesis contradicts the other; i.e. that the states should have different signs. We see then that the overall level of compatibility is increased whenever we find positive weights between units which have states with the same signs, and is decreased in all other cases. Moreover, the degree to which compatibility is increased or decreased depends on the size of the weight and the respective levels of activation.

Fortunately, although there are a large number of possible situations involving positive and negative states and weights, we can measure the contribution to compatibility just by calculating the product of the two states and the weight.

$$s_i s_j w_{\{i,j\}}$$

Note that all three values in this formula may be either positive or negative. But the formula takes care of all possible situations. If s_i and s_j have the same sign and the sign of $w_{\{i,j\}}$ is positive, then the contribution is positive. In all other cases the contribution is negative. Furthermore, the *size* of the contribution always depends on the values in question — which is exactly what we want.

To work out the total level of compatibility we iterate over all the weights summing up the relevant contributions using the following formula.

$$\sum_{i < j} s_i s_j w_{\{i,j\}}$$

The compatibility measure shown above has at least two different names in the connectionist literature. In the PDP workbook [3], compatibility is referred to as *goodness*. In the formulation due to Hopfield [2], *incompatibility* is construed as *global energy* in accordance with the physical analogy mentioned above.

4.1 Energy gaps

When a particular unit in a network changes its state, the global goodness (energy) changes. We can work out by how much the goodness changes fairly easily.

Note that the simplified goodness formula looks at each connection in the network once only. Thus the contribution made by the i 'th unit is just the sum of all the contributions associated with its connections. To calculate unit i 's contribution we work out the sum of all the products of the weights to, and states of, connected units and *then* multiply the result by unit i 's state; i.e. we evaluate

$$s_i \sum_j s_j w_{\{i,j\}}$$

The increase in global goodness produced when the unit i unit changes its state from $s_{1,i}$ to $s_{2,i}$ is then just

$$s_{2,i} - s_{1,i} \sum s_j w_{\{i,j\}}$$

To maximise the overall goodness of the state of the network we should try to put units into states which maximise their contribution. It should be obvious that if

$$\sum s_j w_{\{i,j\}}$$

is positive then the goodness contribution of the i 'th unit will be increased if the unit *increases* its activation level. If the sum is negative then the contribution will be increased if the unit decreases its activation level. Note that this is precisely the processing regime described above. Thus we find that by iteratively increasing the activation level of units whose net input is greater than 0, and decreasing the activation level of units whose net input is less than 0 we will tend to increase global goodness (i.e. decrease global energy).

In Hopfield's 1982 formulation of the current processing regime units can take on only two states (-1 and +1) and the change in the contribution made by a single unit as a result of a change in state is defined as the *energy gap* of the unit.

Hopfield showed that the process whereby we continuously update units so as to minimise their contribution to global energy (which involves putting them in the +1 state if their energy gap is positive and the -1 state otherwise) necessarily reduces global energy. He also showed that it doesn't matter what order we update units in provided that **no unit is ever ignored for more than a finite time** [4, p. 192]. In a later paper Hopfield dealt with the case where units can take on graded activation values [5].

5 The problem of local minima

The Hopfield processing regime is a gradient descent process. At each stage, a change is introduced which is known not to increase the global energy. Like any gradient descent process, this procedure may get stuck in local minima. The problem can be alleviated by allowing units to take on a greater range of states. This increases the size of the energy surface and therefore decreases the a priori probability of falling into a minimum.

6 Accessing schemas

An example of a type of behaviour which can be implemented using this sort of mechanism is **schema-accessing**. In this approach, each unit represents a certain feature in a given situation (or, more technically, represents the hypothesis that a certain feature is present in a given situation) and the weights represent the degree to which one feature suggests the presence of another. In this scenario, turning on the unit representing a particular feature will tend to

evoke a pattern of active features which typically go together; i.e. which form an *archetype* or *schema*.

Toxic fruit example

Imagine that we have a set of hypotheses numbered from 1 to 11, as follows.

```
1: Size_large
2: Flesh_hard
3: Color_green
4: Color_brown
5: Skin_smooth
6: Size_small
7: Color_red
8: Flesh_soft
9: Skin_hairy
10: Toxicity_safe
11: Toxicity_dangerous
```

Each hypothesis specifies a given value for a certain property of an item of fruit. Thus the hypothesis `Skin_smooth` corresponds to the hypothesis **the skin of the fruit is smooth**. The hypotheses are as follows.

We have a priori knowledge which allows us to estimate the degree to which one hypothesis supports another. We configure a network of 11 units such that each unit represents one of the eleven hypotheses and we configure the weights so as to represent the degree to which one hypothesis supports another. The matrix of weights derived is shown in Figure ?. The names of the units correspond to hypotheses in the obvious way; i.e. `u1` correspond to hypothesis 1 and so on. Note how the weights capture relationships between the features. For example the strong weight between `u2` and `u10` represents the fact that hard flesh is typically associated with non-toxicity in this domain. Presented using a Hinton-Sejnowski weight-representation, the weights are as shown in Figure ?. In the figure each large rectangle corresponds to a unit and the boxes inside rectangles correspond — position-wise, size-wise and colour-wise — to its weights. If the unit has a strongly negative weight to the unit whose rectangle appears in, say, the bottom left corner of the display, then there will be a large, filled box in the bottom left corner of the rectangle. If the weight is positive, the box will be large but unfilled. And so on.

missing file `fruit_fields`

6.1 Running the network

Imagine that we clamp the unit representing `Toxicity_dangerous` to have activation 1 and then let the network run. How do the activations of the other units change? We can display the activation changes using a representation

	u1	u2	u3	u4	u5	u6	u7	u8	u9	u10	u11
u11	-0.43	-0.43	-0.14	-0.5	0.43	0.14	-0.14	0.14	-0.71	-0.5	-0.5
u10	0.71	1	-0.43	0.14	0.14	0.14	0.14	-0.14	0.71	-0.5	-0.5
u9	0.14	0.14	-0.14	-0.71	-0.5	-0.14	-0.14	-0.14	-0.5	0.71	-0.71
u8	-0.14	-0.5	-0.14	-0.71	0.14	0.14	-0.14	-0.5	-0.14	-0.14	0.14
u7	-0.14	0.14	-0.5	-0.5	0.14	0.14	-0.5	-0.14	-0.14	0.14	-0.14
u6	-0.5	0.14	-0.14	-0.71	0.43	-0.5	0.14	0.14	-0.14	0.14	0.14
u5	0.14	0.43	-0.43	-0.14	-0.5	0.43	0.14	0.14	-0.5	0.14	0.43
u4	-0.14	-0.14	-0.5	-0.5	-0.14	-0.71	-0.5	-0.71	-0.71	0.14	-0.5
u3	-0.43	-0.43	-0.5	-0.5	-0.43	-0.14	-0.5	-0.14	-0.14	-0.43	-0.14
u2	0.43	-0.5	-0.43	-0.14	0.43	0.14	0.14	-0.5	0.14	1	-0.43
u1	-0.5	0.43	-0.43	-0.14	0.14	-0.5	-0.14	-0.14	0.14	0.71	-0.43

Figure 1:

introduced in [6]. In Figure ? unit names appear on the left. The activation for a unit is shown as a rectangle to the right of the unit's name. The size of the rectangle corresponds to the level of activation in the same way as in the weight representation used above. The position of the rectangle gives the cycle in which the activation level occurred. Rectangles further to the right represent activation levels in later cycles.

missing file toxic_fruit_activations

As we can see, clamping Toxicity_dangerous on tends to produce a certain pattern of activation in which Flesh_soft, Size_small and Skin_hairy are also strongly active. In a sense, clamping the Toxicity_dangerous unit on tends to evoke the toxic fruit schema — it tends to bring out the fact that an item of toxic fruit is likely to be small, soft and hairy.

An example in which the network performs something closer to constraint optimisation is produced if we clamp on the units representing Colour_brown and Flesh_hard. The behaviour produced is shown in Figure ?. The unit representing Toxicity_safe develops a strong, positive level of activation while the one representing Toxicity_dangerous remains inactive. In some sense the network is showing that, other things being equal, a hard, brown fruit is likely to be safe to eat (as well as large and smooth).

missing file pics/hard_fruit_activations

7 Learning methods for Hopfield Nets

The main disadvantage of Hopfield's technique is that it requires the network to be hand-coded. In many cases hand-coding is not a practical proposition. There may be an extremely large number of distinct hypothesis; or the information

concerning evidential support may not be available. Most likely of all, the problem at hand may not naturally translate into a constraint optimisation problem. Ideally, then, we require a method by which we can automatically obtain network configurations which will solve an arbitrary computational problem.

Hinton and Sejnowski have described a procedure called the *Boltzmann Machine Learning procedure* which provides the required functionality for Hopfield nets. The procedure is essentially a *supervised learning* procedure. That is to say, it involves training the network to produce certain behaviour by providing it with examples of that behaviour. The presentation of an example involves clamping an input vector onto the input units and the desired output vector for that input onto the output units. Training involves reinforcing those patterns of activity in the network which are likely to assist the network in producing the desired output when only the input is presented; ie. in producing the desired behaviour.

In practice, the training regime involves reinforcing the weights on links between units which tend to be active simultaneously. It is therefore classed as a type of *Hebbian learning* after Donald Hebb who first proposed the idea that this procedure might be the basic learning mechanism in biological neural networks [7]. The implementation of the procedure is relatively complex and involves computing simultaneity statistics with the network running at *thermal equilibrium*. It has been demonstrated that under this regime, the network is guaranteed to improve its performance, while being necessarily subject to the ubiquitous problem of local minima. Unfortunately, the proof of this result is beyond the scope of the article.

8 Feed-forward networks

There are various problems with the training procedures for Hopfield nets. In particular, the procedure described above is notoriously slow although recent work on a deterministic variant shows how the efficiency of the technique can be radically improved [8]. Due to these problems, most of the work on CC has taken a different approach. In particular, the focus has been on layered, *feed-forward* networks.

In a feed-forward network we have a layer of input units. From these input units we have links leading to a second layer of units. From these units in the second layer we have links leading to units in a third layer, and so on. Eventually we have links leading to a layer of output units. The intermediate layers are called *hidden layers* and their units are *hidden units*. In a strictly layered network all links extend from units in layer n to units in layer $n+1$. There are no links which jump one or more layers. But not all work focusses on the strictly layered net.

Running the network involves setting the activation of the input units. This activation then **feeds forward** through the various layers and eventually arrives at the output units. There are many training procedures for feed-forward nets. However, a substantial proportion of them are supervised learning procedures

involving some kind of error-correction. Typically, the main steps are as follows. First an input is presented. The network is then run and the output produced is read off. This output is compared to the desired output and the weights in the network are adjusted so as to reduce the difference between the actual output and the desired output (ie. the error) in the next iteration. Eventually, if the learning succeeds, the network produces the correct (ie. desired) output for any given input. As with the learning procedure for Hopfield nets, these error-correction procedures are gradient-descent methods and are therefore vulnerable to the problem of local minima.

8.1 The perceptron

One of the simplest applications of CC involving feed-forward nets is the *perceptron*. This application was originally described by Rosenblatt in the early 60s [9] although it is closely related to mathematical procedures with earlier origins. In its simplest form, a perceptron is based on a feed-forward network in which there are n input units and one output unit, see Figure ?. The output unit is a threshold unit; that is to say, it adopts the *on* state (eg. activation = 1) if its net input exceeds some fixed threshold. Otherwise it adopts the *off* state (eg. activation = 0).

`missing file cnnctnst_cmptng_perceptron`

Rosenblatt focussed most of his attention on an application of this architecture in which the input units effectively sample certain pixels of a simulated retina. The goal of learning in such cases was typically that the perceptron should produce a **yes** (ie. an activation of 1 at the output unit) only if the image on the retina was of a certain type; eg. a convex pattern. Rosenblatt used a variant of the error-correction procedure in which weights were reduced/increased depending on the degree to which they had contributed to a correct/incorrect response.

The idea underlying this procedure is very simple. To reduce the output error we need to find out which weights are primarily **responsible** for it and change them accordingly. If the desired output activation is 0 but the actual output activation is 1 we need to reduce large weights connected to active input units. If the desired output activation was 1 and the actual output was 0 we need to increase small weights connected to active input units. And so on. In general, we need to continually reduce each weights contribution to global error. Mathematically speaking we need to iteratively update each weight by an amount proportional to ΔE , where dE is the change in global error and dw is the change in weight w .

Rosenblatt showed that his perceptron learning procedure was capable of producing networks able to recognise various classes of pattern. More importantly he proved that his learning procedure (the *Perceptron Convergence Procedure*) will necessarily find the required set of weights if such a set exists provided that the degree to which a weight is changed in each time step is small enough.

However, in a famous work, Minsky and Papert demonstrated that the perceptron is only capable of learning to distinguish two classes if those classes are *linearly separable* in the input space (ie. if it is possible to find a linear boundary separating one class from the other) [10]. The fact that many problems appear to involve forming input/output mappings in which the input classes are not linearly separable led to a disenchantment with the perceptron and with CC in general.

More recently, however, enthusiasm for feed-forward nets has been increasing. This is due in part to the development of a robust learning procedure for feed-forward networks with *hidden* layers. This procedure is based closely on Rosenblatt's perceptron procedure. But whereas Rosenblatt's procedure was only able to minimise global error in networks with no hidden layers, the generalised procedure could do so in networks with arbitrary many layers. The generalised procedure involves a process in which the initial error values (derived from the differences between the actual and desired outputs) are propagated back through the layers of the network, enabling weights to be adjusted so as to reduce global error.

The first step in this procedure involves computing the error levels for the output units. This is simply achieved by measuring the difference between the actual activation of each output unit and its desired activation (for the given input). We then compute error levels for the units in the layer below. In computing the error level of unit U in layer n we take into account the error level of any unit which U feeds into, the weight on the link between the units and the activation of U. For example, if the error levels of all the units which U feeds into are relatively high, the weights on connecting links are relatively big and the activation of U is relatively high, then we can assign a relatively high error level to U.

It is shown in [11] that this procedure is guaranteed to reduce the global error measure. However, since it is a gradient descent process it is liable to become stuck in local minima. The originators of the procedure claim that in practice it rarely does so. Its resilience in this respect may have something to do with the way in which the procedure is typically applied. Normally, back-propagation is employed in networks with a fairly large number of links. The learning procedure performs gradient descent in weight space and weight space has one dimension for each distinct weight. So in relatively large nets, the gradient descent takes place in a space of high dimensionality. A local minimum only exists if there is a coincident minimum in all dimensions. Thus, the more dimensions, the lower the a priori likelihood of there being a local minimum.

9 Example of back-propagation

In this example we will use back-propagation to derive a network which performs the 4-2-4 encoding task. This task involves taking a vector containing three 1s and one 0, encoding it as a 2-bit binary vector and then decoding it to reconstitute the original vector. We will use a feed-forward network of three

layers as shown in Figure ?F. This net has four input units, two hidden units and four output units. The task is not as easy as it appears since solving it involves being able to form a representation of a 4-bit vector using just two activation values (ie. the activations of the hidden units).

missing file `cnnectnst_cmptng_4-2-4_net`

In this example there are just four possible input vectors. These are

<1 0 0 0>
<0 1 0 0>
<0 0 1 0>
<0 0 0 1>

The target output vectors are identical to the input vectors in every case.

missing file `cnnectnst_cmptng_4-2-4_error`

If we apply back-propagation with the given network and training set using fairly typical values for other parameters of the procedure¹ the procedure reduces the global error relatively fast. Figure ? shows an error profile for one run of back-propagation on this problem. Global error measured in terms of the average difference between target output activations and actual output activations is plotted on the vertical axis. Elapsed time measured in *epochs* is shown along the horizontal axis.² Note that after 250 epochs the global error has fallen quite low. At this point the network is capable of implementing the target behaviour quite closely. In fact the responses of the network at this point are as shown in Figure ?. In learning to produce this behaviour, the network has set weights so that the two hidden units effectively implement a binary encoding of the position of the 1 in the input vector; see Figure ?. The weights on the links from the hidden units to the output units correspond approximately to the four possible encodings involving just two binary values: [0 1], [1 0], [0 0], [1 1]. Since four different cases have to be encoded, a representation using only two binary values is the most compact that can be achieved. Thus in this case we find that back-propagation has learned a configuration of weights which allows for an optimal internal representation.

missing file `cnnectnst_cmptng_4-2-4_fields`

10 Concluding comments

The article has tried to give the reader some flavour of connectionist computational methods. We have looked at two of the better known methods in connectionist computing but we have only considered these in the context of toy

¹We used a learning rate of 0.25 and a momentum of 0.9 for this example.

²An epoch is a unit of time in which all the training inputs are presented once.

```

input [1 0 0 0]
target [1 0 0 0]
output [0.931 0.0002 0.0509 0.0768]
error 0.0575

input [0 1 0 0]
target [0 1 0 0]
output [0.0002 0.9317 0.0545 0.073]
error 0.0569

input [0 0 1 0]
target [0 0 1 0]
output [0.0688 0.0568 0.9166 0.0009]
error 0.0611

input [0 0 0 1]
target [0 0 0 1]
output [0.04 0.0484 0.0002 0.9109]
error 0.0545

```

Figure 2:

problems. Many of the more important issues and outstanding problems have been excluded from the discussion. Furthermore, no account has been taken of some of the more recent CC models, many of which involve dynamic learning architectures (ie. networks which change their architecture dynamically). The reader is referred to [12,13,14] for examples of recent methods.

So what is the prognosis for connectionism in general? Over the past few years there has been an extraordinary increase of activity in connectionist computing. Inevitably, some of this activity has been associated with exaggerated claims and overblown speculations. However, it seems clear that many of the techniques being explored are not simply reinventions of existing statistical and/or mathematical ideas. A major issue is the question of computational efficiency. At present most connectionist work involves the simulation of neural networks on conventional, serial machines. This means that training times for a complex learning task can be prohibitively long. The long term viability of the connectionist paradigm as an engineering enterprise depends to a degree, therefore, on the development and application of parallel computing methods; ie. the further development of hardware implementations of connectionist machinery.

At present the main possibilities for applying CC to problems in CAD lie in the use of architectures such as Hopfield nets for the purposes of solving difficult constraint satisfaction problems. In the case where design constraints can be coded up in the form of numeric weights representing interactions between dif-

ferent design paths, it is possible to find an optimal path by simply running the network to thermal equilibrium. At present, connectionist methods for dealing with structural problems of the kind that are common within CAD are not able to compete directly with traditional computational methods. However, further research may rapidly change the position.

As a final comment, it is worth noting that work on CC seems to raise issues which are too easily ignored in conventional symbol-oriented cognitive science [15]. Overall connectionist computing is an exciting area of research whose main practical applications are slim as yet, but may soon become much more significant. It is to be hoped that research in this area will lead to new insights into computation and learning.

References

- [1] McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5 (pp. 115-33).
- [2] Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79 (pp. 2554-8).
- [3] McClelland, J. and Rumelhart, D. (1988). *Explorations in Parallel Distributed Processing: A handbook of Models, Programs, and Exercises*. Cambridge, Mass.: MIT Press.
- [4] Hinton, G. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40 (pp. 185-234).
- [5] Hopfield, J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, 81 (pp. 3088-3092).
- [6] Rumelhart, D., Smolensky, P., McClelland, J. and Hinton, G. (1986). Schemata and sequential thought processes in PDP models. In D. Rumelhart, J. McClelland and the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructures of Cognition. Vols I and II*. Cambridge, Mass.: MIT Press.
- [7] Hebb, D. (1949). *The Organization of Behavior*. New York: Wiley.
- [8] Hinton, G. (1989). Deterministic boltzmann learning performs steepest descent in weight space. *Neural Computation*, 1 (pp. 143-150).
- [9] Rosenblatt, F. (1962). *Principles of Neurodynamics*. New York: Spartan Books.
- [10] Minsky, M. and Papert, S. (1969). *Perceptrons*. Cambridge, Mass.: MIT Press.

- [11] Rumelhart, D., Hinton, G. and Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, *323* (pp. 533-6).
- [12] Fahlman, S. and Lebiere, C. (1990). The cascade-correlation learning architecture. In D.S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2* (pp. 524-532.). Morgan Kaufmann Publishers, Los Altos CA.
- [13] Mezard, M. and Nadal, J. (1989). Learning in feedforward layered networks: the tiling algorithm. *J. Phys. A: Math. Gen.*, *22* (pp. 2191-2203).
- [14] Frean, M. (1989). *The Upstart Algorithm: A Method for Constructing and Training Feed-Forward Neural Networks*. Edinburgh Physics Preprint 89/479, Dept. of Physics, University of Edinburgh.
- [15] Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3* (pp. 261-283).