Cascaded Back-Propagation on Dynamic Connectionist Networks

Jordan B. Pollack
Computing Research Laboratory
New Mexico State University

ABSTRACT

The Back Propagation algorithm of Rumelhart, Hinton, and Williams (1986) is a powerful learning technique which can adjust weights in connectionist networks composed of multiple layers of perceptron-like units. This paper describes a variation of this technique which is applied to networks with constrained multiplicative connections. Instead of learning the weights to compute a single function, it learns the weights for a network whose outputs are the weights for a network which can then compute multiple functions.

The technique is elucidated by example, and then extended into the realm of sequence learning, as prelude to work on connectionist induction of grammars. Finally, a host of issues regarding this form of computation are raised.

1. Introduction

Most "Connectionist" (Feldman & Ballard, 1982) or "Parallel Distributed Processing" (Rumelhart et. al., 1986b) models use fixed-structure networks, in which the weights are set programmatically or are adjusted slowly by some iterative learning algorithm. The resultant networks are essentially "hard-wired" special-purpose computers that perform some application, like a 10-city traveling Salesman problem (Hopfield & Tank, 1985), past-tense verb conjugation (Rumelhart & McClelland, 1986), text-to-speech processing (Sejnowski & Rosenberg, 1986), or context-free parsing of bounded-length sentences (Fanty, 1985; Selman, 1985). This last application is particularly disturbing because a bounded-length context-free grammar is simply a regular grammar, recognizable by a simple finite-state machine. If connectionism entails a return to pre-Chomskian theories of linguistic capabilities, then it will be in trouble.

One of the major differences between our work in connectionist language processing (Pollack & Waltz, 1982; Waltz & Pollack, 1985) and others (Cottrell, 1985; Fanty, 1985; Selman, 1985) is our use of dynamically changing network structure, i.e., weights that are modified during a computation. Various researchers have seen the need for dynamic connections, including (Feldman, 1982) and (McClelland, 1985), but the resulting systems are very difficult to manage. In the most unconstrained case of a system using multiplicative connections, each weight in a system of n nodes can be a function of the activities of all n nodes, leading to a system with n^3 "parameters" instead of n^2.

But without some form of dynamic connections the generative capacity of connectionist models is suspect. In the past, we have used "normal" computer programs such as a chart parser (Kay, 1973) to dynamically connect our network. This paper outlines steps towards a better way.

A modified form of back-propagation is applied to networks with constrained structures of multiplicative connections and feedback to build systems capable of learning to sequentially process inputs. Using multiplicative connections allows all weights in the system to be dynamically modified for each input.

The technique, called *Cascaded Back-Propagation*, is introduced by comparison to normal back-propagation on feed-forward networks.

2. Back-Propagation

The basic form of the (Rumelhart et. al., 1986a) learning algorithm is as follows. A non-iterative feed-forward network of several layers computes input/output relationships using a continuous version of a perceptron.

Each unit i has an output bounded between 0 and 1. This bounded output is computed by "squashing" its input, x, (a linear combination of weights and other outputs) with the sigmoid function:

$$\Gamma(x) = \frac{1}{1+e^{-x}}$$

which has a derivative (after some algebra) of:

$$\Gamma'(x) = \Gamma(x)(1-\Gamma(x))$$

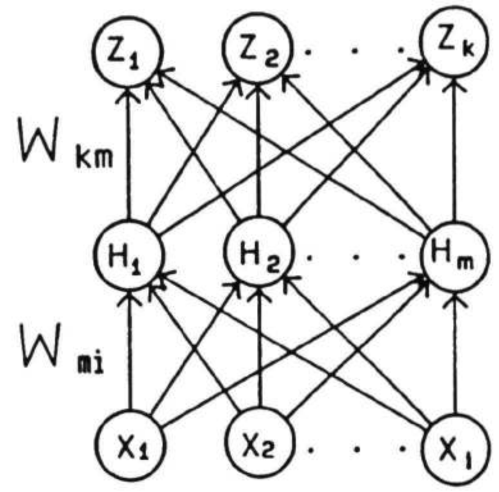

Figure 1:

A simple feed-forward network. Each layer is completely connected to the next. The weights, therefore, are representable by rectangular arrays.

For a simple layered network as shown in Figure 1, this feed-forward computation is as follows:

$$\vec{H}_m = \Gamma(W_{mi} \cdot \vec{X}_i)$$

$$\vec{Z}_k = \Gamma(W_{km} \cdot \vec{H}_m)$$

Where $\vec{X}_i$ is the set of inputs, $\vec{H}_m$ are the outputs of the hidden units, and $\vec{Z}_k$ are the outputs. Back-propagation is given a set of cases consisting of matched pairs of input and desired output vectors. The overall error, E, can be computed as the distance between all desired output vectors $\vec{D}_k^c$ and the actual output vectors computed by the forward-pass, $\vec{Z}_k^c$:

$$E = \sum_c \sum_k (D_k^c - Z_k^c)^2$$

The backward pass works by distributing this error to all the weights in the system. For a particular input/output case, c, this computation is as follows:

$$\frac{\partial E}{\partial \vec{Z}_k} = (\vec{Z}_k - \vec{D}_k)\vec{Z}_k(1 - \vec{Z}_k)$$

$$\frac{\partial E}{\partial \vec{H}_m} = \frac{\partial E}{\partial \vec{Z}_k} \cdot W_{km}\vec{H}_m(1 - \vec{H}_m)$$

$$\frac{\partial E}{\partial W_{km}} = \frac{\partial E}{\partial \vec{Z}_k} \times \vec{H}_m$$

$$\frac{\partial E}{\partial W_{mi}} = \frac{\partial E}{\partial \vec{H}_M} \times \vec{X}_i$$

And by summing the weight errors over all cases and updating each weight by a fraction of its error, μ, plus a fraction, α, of its previous error, ΔW, the algorithm can find a set of weights by gradient descent:

$$W' = W - \mu \frac{\partial E}{\partial W} + \alpha \Delta W$$

$$\Delta W = -\mu \frac{\partial E}{\partial W}$$

A couple of details complete the algorithm. First, initial weights need to be chosen; if all weights are initially 0, there is no way for the system to allocate error, so usually weights are chosen as very small random numbers, say, between ± 0.5. Secondly, the network needs to be "grounded" by adding a "bias" to each hidden and output unit. This amounts to adding another input unit whose output is always 1; the adjustment of the biases then co-occurs with adjustment of all the other weights in the system.

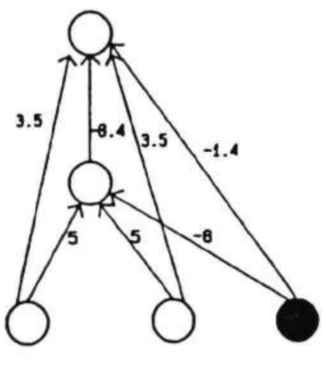

Figure 2:
A simple feedforward network for the exclusive-or problem. Instead of putting biases inside the circles, they are shown as links from a unit with an output of 1.

The simplest test of the algorithm is to learn to compute "Exclusive-or", a function which cannot be learned in a single layer of perceptrons. Figure 2 shows a standard feedforward network on which this algorithm is capable of learning XOR. Using μ=0.5 and α=0.9, back propagation can find these weights usually in a one to two hundred iterations.

3. Cascaded Networks

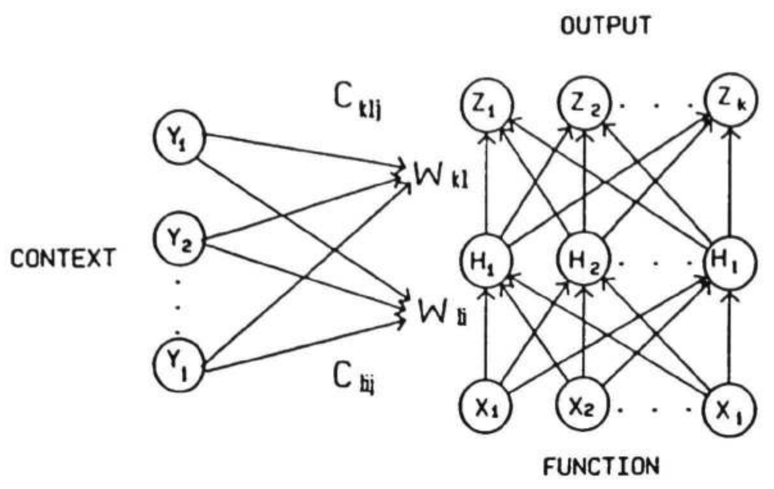

Figure 3:
Cascaded network. A context network with fixed weights runs first and sets variable weights on the function network.

Instead of learning a fixed set of weights for one group of input-output relationships, one network is used to compute some input-output function (the "function network"), and another network (the "context network") is used to compute the weights for the function network (figure 3). By varying the inputs to the context network, the function network can be used to compute various functions.

The forward-pass consists of a forward pass on the context network, which sets the weights on the function network, then a forward pass on the function network:

$$W_{li} = C_{lij} \cdot \vec{Y}_j$$

$$W_{kl} = C_{klj} \cdot \vec{Y}_j$$

$$\vec{H}_l = \Gamma(W_{li} \cdot \vec{X}_i)$$

$$\vec{Z}_k = \Gamma(W_{kl} \cdot \vec{H}_l)$$

Where C_{lij} and C_{klj} represent the fixed weights of the context network, W_{li} and W_{kl} are the varying weights of the function network, $\vec{Y}_j$ are the inputs to the context network, $\vec{X}_i$ are the inputs to the function network, $\vec{H}_l$ are the outputs of the hidden layer, and $\vec{Z}_k$ are the outputs of the function network.

The backward pass consists of computing the errors for the variable weights of the function network, and then using them to compute the errors for the fixed weights:

$$\frac{\partial E}{\partial \vec{Z}_k} = (\vec{Z}_k - \vec{D}_k)\vec{Z}_k(1 - \vec{Z}_k)$$

$$\frac{\partial E}{\partial \vec{H}_l} = \frac{\partial E}{\partial \vec{Z}_k} \cdot W_{kl}\vec{H}_l(1 - \vec{H}_l)$$

$$\frac{\partial E}{\partial W_{kl}} = \frac{\partial E}{\partial \vec{Z}_k} \times \vec{H}_l$$

$$\frac{\partial E}{\partial W_{li}} = \frac{\partial E}{\partial \vec{H}_l} \times \vec{X}_i$$

$$\frac{\partial E}{\partial C_{lij}} = \frac{\partial E}{\partial W_{li}} \times \vec{Y}_j$$

$$\frac{\partial E}{\partial C_{klj}} = \frac{\partial E}{\partial W_{kl}} \times \vec{Y}_j$$

3.1. Exclusive-or Problem

This approach can build networks which runs multiple functions over the same set of units. To compute the exclusive-or function, for example, this amounts to learning the two functions:

$$f(y) = \begin{cases} y & \text{if } x = 0 \\ \neg y & \text{if } x = 1 \end{cases}$$

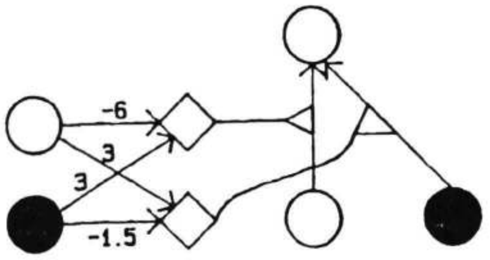

Figure 4:
Cascaded network for the XOR problem. The function network acts as either an inverter or non-inverting buffer depending on the context bit.

Figure 4 shows the cascaded network for the XOR problem. This network needs to learn only 4 weights instead of 7, and, with the learning parameters $\mu = 0.5$ and $\alpha = 0.9$, cascaded back propagation only needs about 30 cycles to learn exclusive-or.

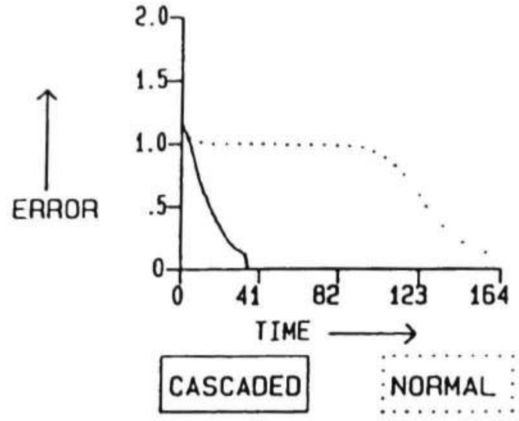

Figure 5:

Typical learning curves for normal versus cascaded XOR network. The horizontal axis represents time as iterations of error propagation, and the vertical represents the global error, E, over all 4 test cases. The algorithms halt when all outputs are within .2 of their desired values.

Figure 5 shows the learning curves for typical runs of back-propagation and cascaded back-propagation for the exclusive-or networks. The number of iterations is represented along the horizontal axis and the global error, E, is represented on the vertical.

3.2. The 4-1 Multiplexor

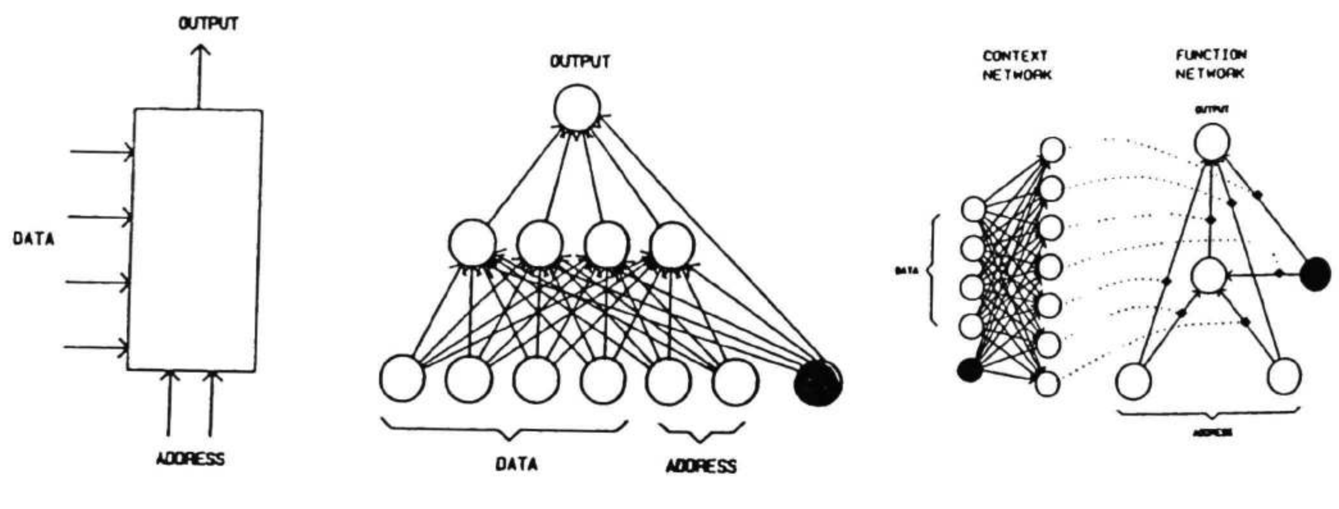

Figure 6:

Three versions of a 4-1 multiplexor. First is the standard block diagram used for logic design; next is a 6-4-1 layered network; and finally, a cascaded network.

Another example is a 4-to-1 multiplexor. A classic logical functional unit used in computer design, it can be thought of as a programmable 2-1 logic function. Figure 6 shows three views of a multiplexor. One of the 4 "data" lines is selected by the values on the 2 address lines. In a normal feed-forward network, there is no distinction between these six inputs, and 4 hidden units are needed to learn all 64 input/output cases. The cascaded network, composed of a single-layer 4-7 network connected to the standard XOR network, essentially learns 5 interacting sets of 7 weights, which produce 16 different sets, one for each logic function.

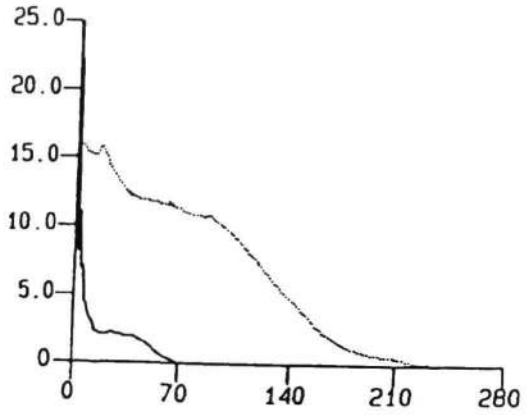

Figure 7:
Typical learning curves for runs of normal versus cascaded back-propagation on the multiplexor problem.

In general, the cascaded solution for the multiplexor problem converges much quicker than the feed-forward solution. Figure 7 shows typical behavior for both solutions with $\mu=0.5$ and $\alpha=0.9$.

4. Sequential Cascaded Networks

When the outputs of the function network are used as inputs to context network, a system can be learned which sequentially processes inputs by dynamically changing the weights in the function network after each input. In parsing terms, it could be said that each word is processed in the context of all the preceding words. And although the number of possible intermediate states are, of course, finite,[1] this system can learn grammars which are bounded in depth, but unbounded in length. The intermediate states must encode various up/down counters. Figure 8 shows a block diagram of a simple sequential cascaded network. Given an initial context, $\vec{Y}_j(0)$, and a sequence of inputs, $\vec{X}_i(t), t=1...n$, the network can compute a sequence of function output/context input vectors, $\vec{Y}_j(t), t=1...n$ by dynamically changing the set of weights, $W_{kl}(t)$ and $W_{jl}(t)$:

1. Unless it is assume that the outputs are true analog values or rational numbers.

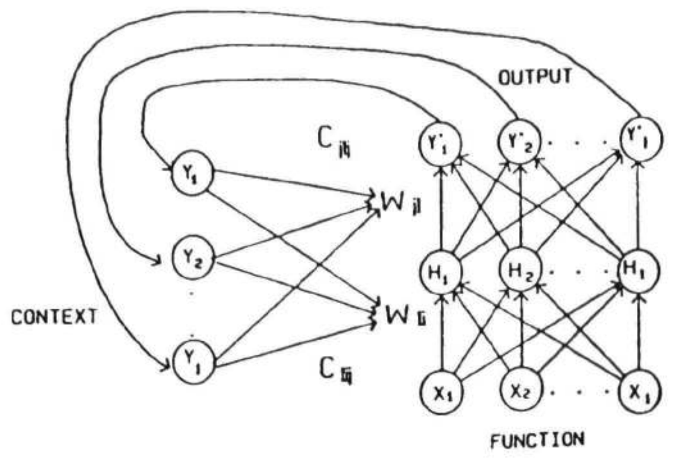

Figure 8:
The outputs of the function network are used as the next inputs to the context network, yielding a system whose function varies over time.

$$W_{li}(t) = C_{lij} \cdot \vec{Y}_j(t-1)$$

$$W_{jl}(t) = C_{jlj} \cdot \vec{Y}_j(t-1)$$

$$\vec{H}_l(t) = \Gamma(W_{li}(t) \cdot \vec{X}_i(t))$$

$$\vec{Y}_j(t) = \Gamma(W_{jl}(t) \cdot \vec{H}_l(t))$$

The error correction phase can be applied to just the final input:

$$\frac{\partial E}{\partial \vec{Y}_j(n)} = (\vec{Y}_j(n) - \vec{D}_j)\,\vec{Y}_j(n)(1 - \vec{Y}_j(n))$$

$$\frac{\partial E}{\partial \vec{H}_l(n)} = \frac{\partial E}{\partial \vec{Y}_j(n)} \cdot W_{kl}(n)\vec{H}_l(n)(1 - \vec{H}_l(n))$$

$$\frac{\partial E}{\partial W_{kl}(n)} = \frac{\partial E}{\partial \vec{Y}_j(n)} \times \vec{H}_l(n)$$

$$\frac{\partial E}{\partial W_{li}(n)} = \frac{\partial E}{\partial \vec{H}_l(n)} \times \vec{X}_i(n)$$

$$\frac{\partial E}{\partial C_{lij}} = \frac{\partial E}{\partial W_{li}(n)} \times \vec{Y}_j(n-1)$$

$$\frac{\partial E}{\partial C_{klj}} = \frac{\partial E}{\partial W_{kl}(n)} \times \vec{Y}_j(n-1)$$

where $\vec{D}_j$ is the desired output for a particular sequence.

4.1. Learning Parity

When Exclusive-or is generalized to more than 2 inputs, it becomes the parity problem, to determine whether a boolean string has an odd or even number of 1's in it. This problem was discussed at length, both by (Minsky & Papert, 1969), as a hard problem for perceptrons and by (Rumelhart et. al., 1986a) as a test case for back-propagation.

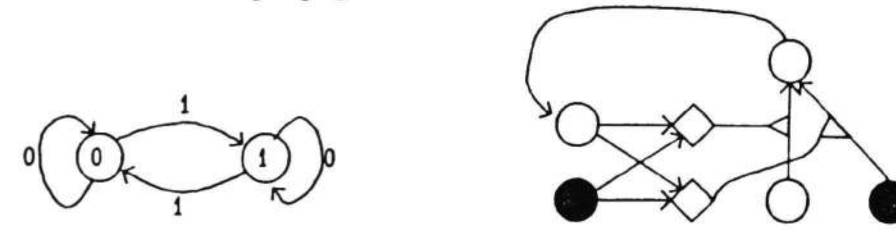

Figure 9:
A simple 2-state machine is shown on the left, and a sequential cascaded network is shown on the right.

A problem for normal back-propagation is that the parity problem of size K requires K hidden units to work, so a system which learned to determine parity of 5 bits would not work for 6. This problem can be overcome by "going sequential", using the cascaded exclusive-or network with feedback between the output of the function network and the input to the context network. This network, and the corresponding small finite-state machine are shown in figure 9.

One problem with the cascaded network approach is that if the system is trained to within .2 of the solution for each training case, the weights "fuzz out" for longer tests than the ones given. There are several solutions to this. The simplest one is to put a truncating filter between the function output and context input which converts outputs above 0.8 to 1 and below 0.2 to 0. Other possible solutions include more complicated filters such as an auto-associative memory or other relaxation system which corrects fuzzy states..

4.2. Parenthesis Balancing

Unfortunately, parity is very unnatural and extremely finite state "language". A real solution to a temporal credit assignment problem with application to language processing is not served by learning such finite systems as parity or even 6-letter sequence completion as used by (Rumelhart et. al., 1986a) to demonstrate recurrent networks.

A connectionist network at least should be able to learn a context-free language from example in order to claim any service to language processing. Accordingly, experiments have been performed in learning the second simplest context-free language known to man: Parenthesis balancing.[2] We have successfully used a sequential cascaded network for parenthesis balancing consisting of of a layered 1-3-2 function network and a 2-14 context net. The input is either 1 or 0 for left and right parentheses and one output signifies grammaticality of the prefix and the other works as a stack, by shifting outputs from 1 to .5 to .25 as more left parentheses are input.

4.3. Other grammars

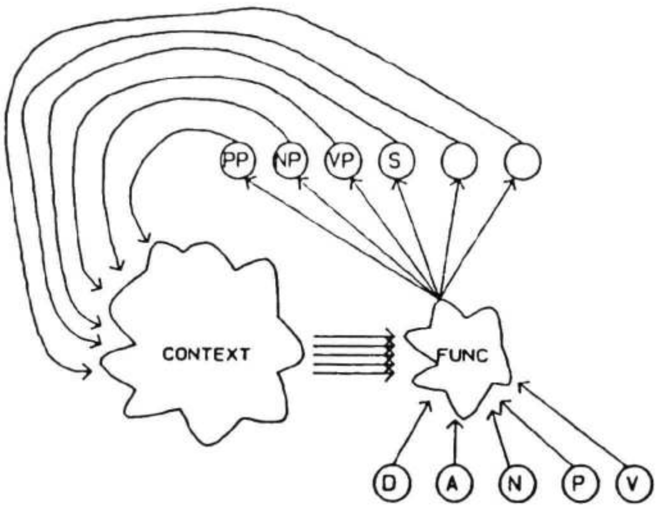

Figure 10:

A sequential cascaded network for parsing. The current state is tied back into the context for the next input, and the unlabeled units would develop necessary features.

Using a sequential cascaded network engenders a time-distributed representation of a parse-tree. For example if a network were used as shown in figure 10, with lexical categories for input and phrase markers for output and context, a system could be developed that, for a simple declarative sentence like "The fat man ate the spaghetti with sauce", could have time-varying outputs which implicitly code its parse tree.

The following table shows the outputs of the phrase-marker units over time. Each contiguous group of "on" states could be interpreted as a single node of a tree, with interval-inclusion determining dominance.

2. The simplest context free grammar is described by the regular expression $a^n b^n$.

S	*	*	*	*	*	*	*	*
VP				*	*	*	*	*
PP							*	*
NP	*	*	*		*	*		*

There are several problems with this representation, the main one being that self-recursive categories are not recoverable. One possibility which is currently being examined is the combination of this learning technique with the representational assumption that outputs can have arbitrary fractional resolution used as stack. Experimentation with representing and learning larger grammars of this type has just begun; clearly more work need be done.[3]

5. Discussion

Cascaded networks do not solve everything, unfortunately. They are basically a very constrained type of network with multiplicative links. This algorithm, being a variation of back-propagation, does not solve its inherent problems. For example, it can still get stuck in local minima, and the exact topology needed to solve a particular problem must be defined beforehand.

But the combination of faster convergence and the computational power engendered by multiplicative connections make cascaded back-propagation a useful technique for connectionist modeling and worthy of further study. Some issues for further discussion are presented below.

5.1. Why is it Faster?

For both the exclusive-or and multiplexor problems given above, as well as various other problems we have experimented with, cascaded back-propagation converges on solutions significantly faster than normal back-propagation. We think this is due, essentially, to the well-known algorithmic technique of "divide-and-conquer". By breaking the solution to exclusive-or into two simpler problems (i.e. an inverting and non-inverting buffer) or the multiplexor into 16 smaller problems (i.e. each 2-1 boolean function), we reduce the amount of work involved tremendously.

Consider running normal back-propagation multiple times, once for each simple problem on the function net, saving the discovered weights, and then once for mapping the context inputs to to these weights. If both nets are capable of learning their functions, then this scheme will work, and the number of iterations needed will be the sum of all the smaller cases.

But when we learn all these subproblems at the same time, the number of iterations will be related to the hardest subproblem to learn. Thus for the exclusive-or problem, the number of iterations we need is related to how hard it is to learn to invert (i.e. even a *perceptron* can do this), and for the multiplexor problem the number of iterations needed will be related to how hard it is to learn normal exclusive-or or equivalence.

Furthermore, a particular solution to a subproblem found by back-propagation is a discrete point in "weight space", somewhere along the edge of a region of good solutions. Running all subproblems first makes the context mapping problem more difficult by adding this unnecessary edge constraint; merging the subproblems and mapping problem removes this constraint and

3. One learning trial may be considered a success, however: We inadvertently used a training set with a simple principle of grammaticality -- the system discovered that all grammatical sentences were of length $0 \, mod \, 3$!

allows each subproblem solution to be anywhere in its region.

5.2. Relation to Sigma-Pi Units

Williams (1986) has classified various activation functions for connectionist models. The ultimate function for combining inputs to a unit are called "Sigma-Pi" functions, which linearly combined multiplied subsets of inputs. So for n inputs, $x_1, \ldots, x_n$, a unit with j weights may provide as output:

$$\sum_{S_j \in P} \prod_{i \in S_j} x_i$$

Where P is the set of all subsets of $\{1,\ldots,n\}$. This is the ultimate in combining functions because when $j=2^n$ a single unit can implement a general polynomial. The down side is that having 2^n weights associated with a single unit is the combinatorial *brut existant*.

As far as classification, however, almost any multiplicative connection system is a special case of Sigma-Pi. For example, the gating activation function described by (Hinton, 1981) uses $j=\frac{n}{2}$ weights, where n inputs are separated into $\frac{n}{2}$ pairs which are multiplied and combined. A cascaded network can be also seen as a special case, where the n inputs are broken into 2 sets whose elements are multipled in pairs with $j=\frac{n^2}{4}$ weights.

5.3. Single-layered Context Networks

In the examples given in the paper, single-layered context networks were used. This construction will work only if the good regions in weight space for each subproblem can be linearly composed with respect to the context inputs. That a single-layer context works with the exclusive-or problem is obvious; that it worked for the multiplexor is surprising. For harder problems, it may turn out that more hidden units are needed in the function network to provide the flexibility for this kind of context network. On the other hand, if back-propagation works, there is no constraint that the context network has to be single-layered.

6. Conclusion: A Universal Neural Network?

Consider the notion of a Universal Turing Machine. A very simple construction which, when presented with a description of any other Turing Machine and its initial state, runs a simulation of that TM to completion. This is similar to a virtual machine emulator or programming language interpreter running on a normal computer. One difference is that because of the random-access property of a computer versus the serial tape access of a UTM, the normal computer runs simulations much faster. For each simulated operation of the TM, the UTM may have to step from one end of its tape to the other. For a program and tape of size n this amounts to about an n^2, or *polynomial* simulation time. A programmed interpreter, on the other hand, has to look up an operation in a table and call a simple routine which updates the state of the interpreted system. Assuming each simulated operation takes about k machine instructions, then the simulation takes kn, or *linear* time. This efficiency advantage is why no modern computers are built like Turing machines.

Consider, finally, an extended cascaded network where the context network is presented a description of a machine and produces the set of weights for the function network. The function network can then run at full "neural speed". If the context network takes a constant time, k, to do its computation, then this simulation runs in $k+n$, or *constant* time.

The point of all this is to solve a conundrum for connectionists: When attempting to model a high-level cognitive domain one quickly realizes the folly of equating a neuron with an element of that domain. Neurons are arrayed in a fixed network, and only die as time goes by. If the memory of your grandmother were localized to a single neuron, and that neuron failed, you would forget her.

One backup position (which this author has resorted to occasionally) is something like the following: The units in my system are not really neurons, but a elements of a higher level system which are somehow simulated by neurons. The problem for this position has been that simulation takes time, and given the finite number of cycles available for "real-time" cognition, i.e. 10-100 cycles, there isn't any time for the simulation to take place.

If a universal neural network could really run a simulation of a neural network in constant time, then the backup position becomes viable -- the units in the higher-level system are temporarily run on neurons at neural speed. Cascaded networks are a first step in this direction.

7. References

Cottrell, G. W. (1985). Connectionist Parsing. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society.* Irvine, CA.

Fanty, M. (1985). Context-free parsing in Connectionist Networks. TR174, Rochester, N.Y.: University of Rochester, Computer Science Department.

Feldman, J. A. (1982). Dynamic Connections in Neural Networks. *Biological Cybernetics, 46,* 27-39.

Feldman, J. A. & Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science, 6,* 205-254.

Hinton, G. E. (1981). A Parallel Computation that Assigns Canonical Object-Based Frames of Reference. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* Vancouver, B.C., 683-685.

Hopfield, J. J. & Tank, D. W. (1985). 'Neural' computation of decisions in optimization problems. *Biological Cybernetics, 52.*

Kay, M. (1973). The MIND System. In Rustin, (Ed.), *Natural Language Processing.* New York: Algorithmics Press.

McClelland, J. L. (1985). Putting Knowledge in its Place. *Cognitive Science, 9,* 113-146.

Minsky, M. & Papert, S. (1969). *Perceptrons.* Cambridge, MA: MIT Press.

Pollack, J. B. & Waltz, D. L. (1982). Natural Language Processing Using Spreading Activation and Lateral Inhibition. In *Proceedings of the Fourth Annual Cognitive Science Conference.* Ann Arbor, MI, 50-53.

Rumelhart, D. E. & McClelland, J. L. (1986). On Learning the Past Tenses of English Verbs. In J. L. McClelland, D. E. Rumelhart & the PDP research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition,* Vol. 2. Cambridge: MIT Press.

Rumelhart, D. E., Hinton, G. & Williams, R. (1986). Learning Internal Representations through Error Propagation. In D. E. Rumelhart, J. L. McClelland & the PDP research Group, (Eds.), *Parallel Distributed Processing: Experiments in the Microstructure of Cognition,* Vol. 1. Cambridge: MIT Press.

Rumelhart, D. E., McClelland, J. L. & Group, the PDP research (1986). In *Parallel Distributed Processing: Experiments in the Microstructure of Cognition,* Vol. 1. Cambridge: MIT Press.

Sejnowski, T. J. & Rosenberg, C. R. (1986). NETtalk: A parallel network that learns to read aloud. JHU/EECS-86/01: The Johns Hopkins University, Electrical Engineering and Computer Science Department.

Selman, B. (1985). Rule-Based Processing in a Connectionist System for Natural Language Understanding. CSRI-168, Toronto, Canada: University of Toronto, Computer Systems Research Institute.

Waltz, D. L. & Pollack, J. B. (1985). Massively Parallel Parsing: A strongly interactive model of Natural Language Interpretation. *Cognitive Science, 9*, 51-74.