Learning Simple Arithmetic Procedures

Garrison W. Cottrell and Fu-Sheng Tsung
Department of Computer Science and Engineering
Institute for Cognitive Science
University of California, San Diego.

Abstract

Two types of simple recurrent networks (Jordan, 1986; Elman, 1988) were trained and compared on the task of adding two multi-digit numbers. Results showed that: (1) A manipulation of the training environment, called *Combined Subset Training* (CST), was found to be necessary to learn the large set of patterns used; (2) if the networks are viewed as learning simple programming constructs such as conditional branches, while-loops and sequences, then there is a clear way to demonstrate a capacity difference between the two types of networks studied. In particular, we found that there are programs that one type of network can perform that the other cannot. Finally, an analysis of the dynamics of one of the networks is described.

Introduction

One major criticism of artificial neural networks is that there is no obvious method for doing sequential, symbolic processing. Rumelhart, Smolensky, McClelland & Hinton (1986) proposed that symbolic processing may be achieved by **(1)** creating physical representations of the problem, **(2)** processing the representations via pattern association, and **(3)** recording the result of the processing in the physical representation. The example they use is the problem of adding two three digit numbers. First the two numbers are written down in a standard format as on the left:

```
                        1
     327               327
     865               865
     -----             -----
                         2
```

This is now a pattern recognition problem, with the result being recorded by an action, i.e., writing down the sum of the rightmost column as on the right above. This presents a new pattern, which triggers the writing of a carry and the process repeats. Similarly, they claim that a complex logical problem is solved by breaking it down into simpler problems and applying the above procedure repeatedly. We were interested in just what was involved in implementing the above description, especially when a memory load is added by not explicitly recording the carry.

In order to have a PDP network do sequences of actions, it has to have some way of knowing "where" it is in the sequence. One way of accomplishing this is by explicitly having discrete states in the unit functions which change over time (Feldman & Ballard, 1982). An alternative is to have *recurrence* in the network, so that the state of the network is reflected in the activation levels of the units. We adopt the latter approach, following the work of Jordan (1986) and Elman (1988). Both of these approaches are restricted extensions of the basic feed-forward network used in most back propagation experiments (Rumelhart, Hinton, & Williams, 1986) that nevertheless still allow the use of back propagation learning.

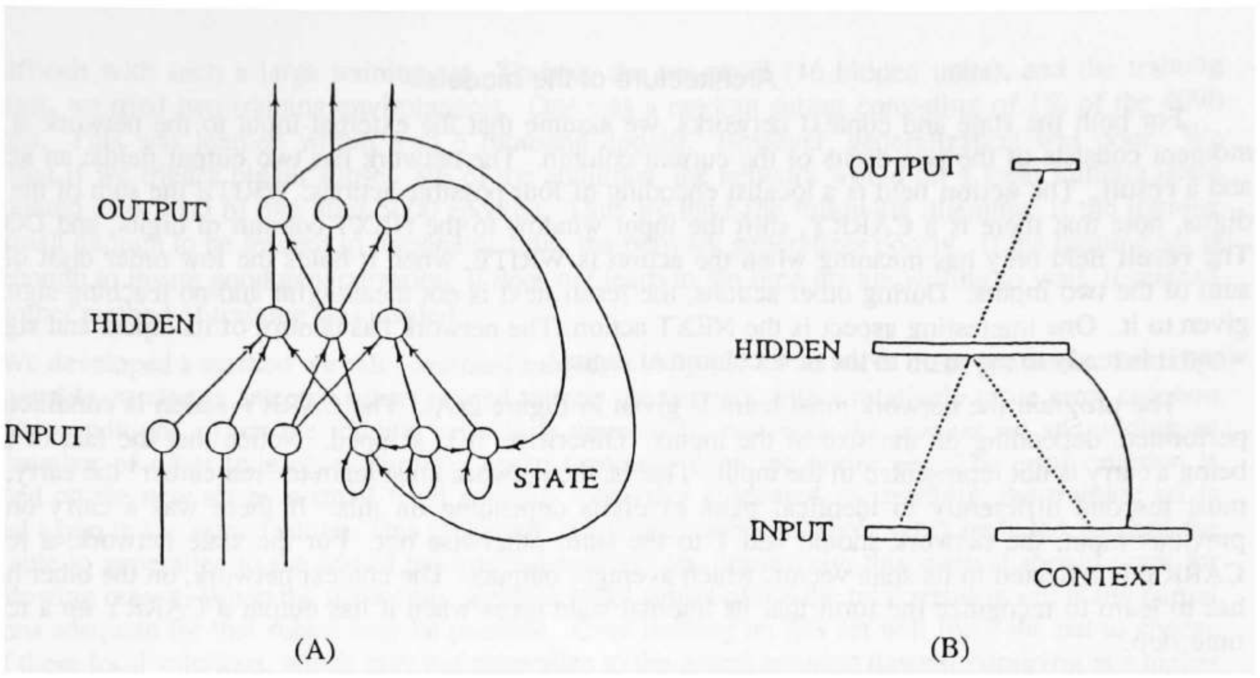

Figure 1. (a) Jordan's recurrent network. The outputs are linearly summed over time in the *state* vector, lower right. (b) Elman's network. The context vector (lower right) is a copy of the hidden units from the previous time step.

In Jordan's approach (see Figure 1(a)), the output vector of the network is linearly averaged into a *state* vector (the same length as the output), which is given to the network as part of the input. We will call these networks "state" networks. The state vector at time t becomes some proportion (*mu*) of its value at time t-1, plus the output vector at time t-1. Thus the network has an exponentially decaying representation of its output history. The other input is called the *plan vector*, that is, an arbitrary representation of the sequence to be produced. This remains constant throughout the processing. State networks can be trained to produce nearly arbitrary sequences.

In Elman's approach (see Figure 1(b)), the hidden unit activations at time t-1 are copied into a *context* vector, which is given as input to the network at time t. This is equivalent to having the hidden units be completely recurrently connected, and back propagating one step in time along the recurrent links. We will call these networks "context" networks. Context networks are typically used to predict their next input, which causes them to represent structural regularities in their environment.

Thus these architectures have typically been applied to very different tasks: The state networks have been used to learn to produce sequences, using a fixed-input plan; the context networks have been used to recognize structural regularities in their input. Hence no comparison of their power has been done. In the following, we apply them both to the same problem: Learning a simple arithmetic procedure. This allows comparison of the two network types. We find that there are procedures that one can perform that the other cannot.

We chose multi-column addition as our symbolic processing task because although it is a simple problem, it is nontrivial for parallel distributed processing (PDP) networks because it involves control processes such as sequential processing and looping that are not traditional PDP tasks. Furthermore, it is an interesting paradigm for generalization, since we can only train the network on a finite set of examples, while there are an infinite number of possible cases. Hence the network must learn the implicit, underlying rule of addition.

Architecture of the models

For both the state and context networks, we assume that the external input to the network at any moment consists of the two digits of the current column. The network has two output fields: an **action** and a **result**. The **action** field is a localist encoding of four possible actions: WRITE the sum of the two digits, note that there is a CARRY, shift the input window to the NEXT column of digits, and DONE. The **result** field only has meaning when the action is WRITE, when it holds the low order digit of the sum of the two inputs. During other actions, the result field is not meaningful and no teaching signal is given to it. One interesting aspect is the NEXT action: The network has control of its inputs and signals when it is ready to move on to the next column of digits.

The program the network must learn is given in Figure 2(A). The CARRY action is conditionally performed, depending on the size of the inputs. Otherwise, it is skipped. Notice that the fact of there being a carry is not represented in the input. That is, the network must learn to "remember" the carry, and must respond differently to identical pairs of digits depending on this. If there was a carry on the previous input, the network should add 1 to the sum, otherwise not. For the state network, a recent CARRY is reflected in its state vector, which averages outputs. The context network, on the other hand, has to learn to recognize the form that its internal state takes when it has output a CARRY on a recent time step.

To reduce the number of basic additions to be learned, we used base 4 instead of decimal. There are thus 16 basic associations for additions, plus the other program elements. The carry complicates the situation, since the net has to respond to each pair differently in the presence of a carry. Worse yet, since the state or context vectors record processing history, the network has essentially an infinite set of unique inputs. Since most of this is irrelevant, one thing the network must learn is to *ignore* its distant history.

Simulations

Training Strategy

The goal is to get the network to learn the addition process for an arbitrary number of digits. Immediate questions are: How to pick a training set? What makes a good one? How many examples are enough for the net to generalize? We somewhat arbitrarily decided to train the network on additions with addends of up to three digits. This set contains all the canonical situations, and therefore should be enough. However, there are 4096 additions (including all 1-digit, 2-digit, 3-digit combinations), and when translated to the network output sequences, there are more than 30,000 individual patterns. Learning is

```
while not done do
begin
    output(WRITE, low_order_digit);
    if sum>radix then
        output(CARRY, ???);
    output(NEXT, ???);
end
if carry_on_previous_input then
    output(WRITE, '01');
output(DONE, ???);
```

(A)

```
while not done do
begin
    output(WRITE, low_order_digit);
    output(NEXT, ???);
    if sum(previous_input)>radix then
        output(CARRY, ???);
end
if carry_on_previous_input then
    output(WRITE, '01');
output(DONE, ???);
```

(B)

Figure 2. (A) The "program" the network learns. There are two output fields, and **action** field and a **result** field. For most outputs, the **result** field is not trained ("???" in the figure). (B) Modified program.

very difficult with such a large training set. To keep the net small (16 hidden units), and the training fairly fast, we tried two training environments. One was a random subset consisting of 1% of the 4096 additions. This was learned within 3 to 5 thousand epochs. However, generalization was poor. We found that if we tried a bigger subset, 8% of the additions, the network would hit a local minima (total sum squared error (tss) of 346 after 5000 epochs). Thus we have the following dilemma: If the training set is small enough to be learned in reasonable time, the network generalizes poorly. If the training set is large enough to insure good generalization, it does not learn in the time we are willing to wait. It seemed that another method of training was needed.

We developed a method we call *combined subset training* (CST)[1] to solve these problems. Initially, a manageable, randomly selected subset is used to train the network with a relatively loose error criterion (stopping condition). Then the training set size is doubled by *retaining the current set* and adding an equal number of other training examples, chosen randomly from the entire set. The error criterion is tightened on the new set by a small fixed amount. Once this is learned successfully, the training set is doubled again in the same fashion. This procedure is repeated until the whole set is included, or until the net is able to generalize to the rest of the original training set. Intuitively, this method should work for the following reason: When the net is only seeing a small subset of the the total training set, many partial solutions adequate for that subset may be possible. Over-training on this set will force the net to choose one of these local solutions, which may not generalize to the global solution desired. Stopping at a higher error criterion leaves the options open by preventing the network from diving too deeply into local minima.

For this experiment, we picked 1% (of the 4096 additions) as our initial set. The networks (both types) are trained to a tss of 1.0, and the training set is doubled to 2% of the total. The total sum squared error (tss) jumps up initially at the introduction of the new examples, but not as high as with the starting weights (see Figure 3a). This shows that the network is already generalizing to some extent. The same behavior was observed when we doubled to 4% and 8%, except that each time, the peak of the error jump is less than the peak before it. At 8% of the training set (close to 3000 individual patterns), we found that

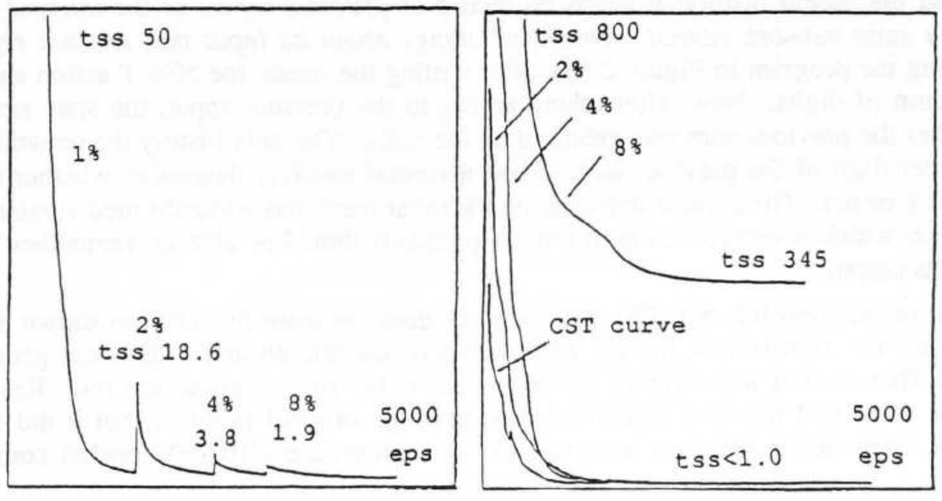

Figure 3. (a) Error curve from combined subset training on the context net. Each jump in tss is where training sets are doubled. Beginning tss: 348.5; final tss: 0.399. (b) Comparison of CST to fixed set training for 2%, 4% and 8% subsets (the y axis is much higher than in (a)).

[1]We report on this method at more length in Tsung & Cottrell (1989).

the net generalizes very well to the rest of the 4096 additions.

In Figure 3(b), we compare the CST procedure outlined above with training on fixed subsets of size 2%, 4% and 8%. Notice that the CST curve falls under all of the others. More to the point, the 8% curve appears to have reached a local minima at tss 346, while the networks trained on fixed subsets of 2% and 4% do not generalize well to the rest of the patterns (data not shown). This illustrates the dilemma stated above: When training on *fixed* subsets, if the training set is small enough for the network to reach criterion, then it doesn't generalize well. With larger training sets, the network does not learn.

We then tested the CST-trained network on longer additions. Even though the network was only trained on additions of up to 3 digits, on tests of 100 longer additions, the miss rate is only 10%. We then trained the net on a part of the test set of longer additions. In general, we found that (see Tsung & Cottrell 1989):

(A) Training on a small part of the test set corrects performance on the rest. This suggests that there only one, or few, classes of errors the net is prone to make.

(B) The network learned to correct the mistakes quickly (within tens of epochs of further training). Furthermore, the extra training does not upset the performance on the 3-digit additions.

(C) Further generalization tests showed very little error, with a miss rate of less than 1%.

From the patterns of error behavior and the observations listed above, it is clear that the network has learned the task, needing only small refinements.

Differences between state and context nets

Both state and context networks behaved similarly on this task. We may ask the question: Is there some task that one can do, and not the other? The answer is yes, and a simple example is found by interchanging two lines in the program the network must learn. Instead of the original sequence of "write result-carry-next", the net is trained to output "write result-next-carry", as in Figure 2 (B). A state network should not be able to solve this problem. This is because the state network has access to only the current input and the *output* history; it keeps no record of previous inputs or the *internal* states of the system. Thus, *a state network cannot "remember" things about its input that are not reflected in its output*. Following the program in Figure 2 (B), after writing the result, the NEXT action shifts the input to the next column of digits. Now, after losing access to the previous input, the state network has to determine whether the previous sum was greater than the radix. The only history the network has reflects only the low order digit of the previous sum. Thus it cannot possibly determine whether the next step should be CARRY or not. The context network, on the other hand, has a transformed version of the input at the hidden layer which is recycled at each time step, thus it should be able to "remember" input that is not reflected in its output.

Simulation results bear this out. The state network does not learn this task, as shown in Figure 4. It achieved a low tss for each subset, but the error curve is not smooth and it does not generalize to the doubled subset. That is, it is memorizing the sequences rather than learning the task. Results with the context network show that this is a harder problem than the original problem, but it did learn it. The context network takes about twice as long to learn this procedure (~10,000 epochs) compared to the previous version.

Initial analysis of the internal representation

To look at the *dynamics* of the network as it moves through the problem, we found the principal components of the 16 hidden unit activations over time, as the context network (using the program in Figure 2 (A)) processed 10 additions from one of the generalization test sets. This analysis gives the directions of highest variance of the hidden unit activations over time. Basically, we can think of this as finding a new coordinate space for the hidden unit vectors, where the coordinate vectors are ordered in terms of how much "action" occurs along each one.

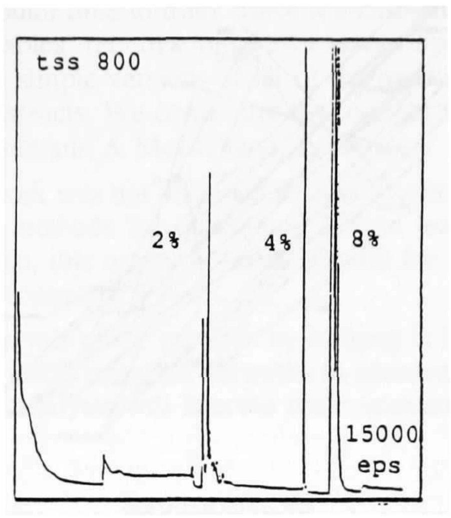

Figure 4. Learning program from Figure 2 (B) with the state network. CST doubling is indicated. The net is learning local solutions, and is unable to generalize.

In Figure 5, we show the projection of the hidden unit vectors onto the plane of the first two principal components as the network is doing a 30 step addition. Each point is labeled by the action that is produced on the output and the step in the entire computation. This shows how the network moves through its internal states as it processes the input. There are several things to notice here. Basically, WRITE result actions (labeled R#) are generally in the right half of the space, NEXTs and CARRYs are

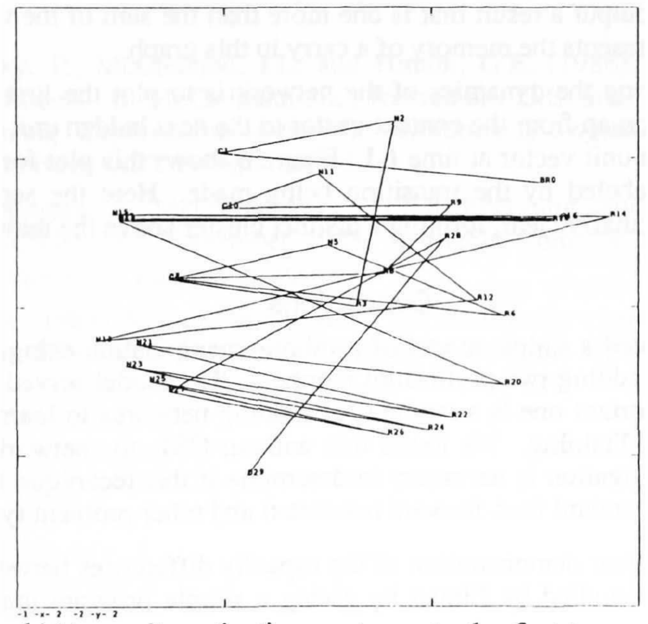

Figure 5. Projections of the hidden units activation vector onto the first two principal components. The number in the point labels corresponds to the step in this addition. R: RESULT, C: CARRY, N: NEXT, D: DONE.

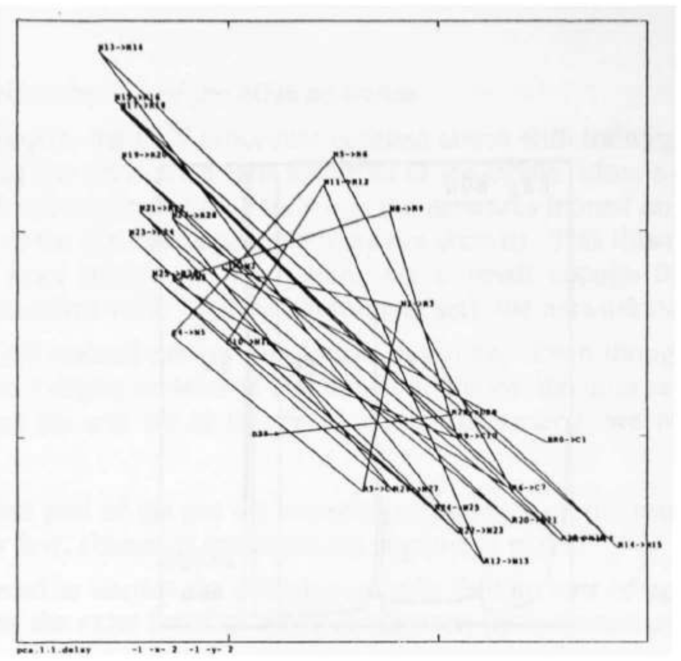

Figure 6. Projection of the hidden unit activation vector onto principal component 1 plotted against itself one time step later.

in the left. Second, in general, the second component is correlated with overall time within this problem. It is interesting that the network represents absolute time even though it is unnecessary for solving the problem. The most striking result that emerges from this analysis is that on the first principal component (the **x** axis), the network is distinguishing between a NEXT that follows a CARRY, versus one that follows a WRITE. All of the NEXTs following a CARRY are greater than 0 on this axis, all of those following a WRITE are less than 0. The significance of this is that following a NEXT that follows a CARRY, the network must output a result that is one more than the sum of the two inputs. We can thus see the internal state that represents the memory of a carry in this graph.

A second way of viewing the dynamics of the network is to plot the first principal component at time **t** vs. **t+1**. This gives the map from the context vector to the next hidden unit vector, since the context vector at time **t** is the hidden unit vector at time **t-1**. Figure 6 shows this plot for the same problem as in Figure 5. Here points are labeled by the transition being made. Here the separation of the NEXT's following a CARRY is particularly clear, forming a distinct cluster above the main diagonal of the graph.

Conclusions

In this paper we presented a simple model of symbolic manipulation using a connectionist network that learned a procedure for adding two multi-digit numbers. The model served as a catalyst for several other results. The most important one is a method for training networks to learn large training environments via Combined Subset Training. We found that without CST, the networks we studied could not learn the task. Further investigation is necessary to determine if this technique is suitable for other network architectures (such as standard feed-forward networks) and other problem types.

The second result is a clear demonstration of the capacity differences between the type of networks studied by Jordan and those studied by Elman by giving a simple program that one can learn that the other cannot. The basic notion may be stated as follows: Networks with only output histories cannot remember things about their input that are not reflected in their output. This is perfectly clear now; it was not when we started this research.

The third result is a demonstration that networks of this type can learn simple programming constructs that are not nested. In particular, these nets can do simple sequencing, looping, and branching.

Also, values necessary for future processing can be stored over short periods by the context network. Other recurrent network models are more powerful in this regard (Williams & Zipser, 1988), but require a prohibitively large amount of computer time to train. Since remembering a bit takes a long time to learn, this suggests that memory for "variables" requires initial structures subserving this function that are easily refined by learning. We thus have simple versions of all of the mechanisms for a universal computer--except the ability to nest these constructs. We conjecture that nesting will not be able to be carried very deeply (cf. Servan-Schreiber, Cleeremans, & McClelland, 1988).

Fourth, even though the network was not designed to be a psychological model of human learning, it may provide some insight into methods for optimizing human learning in terms of structuring the problem sets of addition facts. Also, this model is fertile ground for exploring other aspects of human procedural learning and symbolic processing.

Finally we have begun an analysis of the network by looking at its state space graph. This type of analysis is necessary when we are using recurrent networks to observe the dynamics of the system. We expect that the use of this kind of analysis will become more commonplace as more researchers study recurrent networks.

REFERENCES

Elman, J. (1988) Finding structure in time. Technical Report 8801, Center for Research in Language, University of California, San Diego, La Jolla, California.

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

Jordan, M. (1986) Serial order: A parallel distributed processing approach. Technical Report 8604, Institute for Cognitive Science, University of California, San Diego, La Jolla, California.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, & the PDP Research Group, *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1. Foundations.* Cambridge: MIT Press/Bradford. Books.

Rumelhart, D.E., Smolensky, P., McClelland, J.L. and Hinton, G.E. (1986) Schemata and sequential thought processes in PDP models. In McClelland, J.L., Rumelhart, D.E. and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2: Psychological and Biological Models.* Cambridge: MIT Press/Bradford.

Servan-Schreiber, D., Cleeremans, A., McClelland, J.L. (1988) Encoding sequential structure in simple recurrent networks. CMU-CS-88-183, November 1988. Available from the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Williams, R. and Zipser, D. (1988) A learning algorithm for continually running fully recurrent neural networks. Technical Report ICS-8805. Institute of Cognitive Science, University of California, San Diego, La Jolla, California.