This article was downloaded by: *[CDL Journals Account]*
On: *30 October 2008*
Access details: *Access Details: [subscription number 786945878]*
Publisher *Taylor & Francis*
Informa Ltd Registered in England and Wales Registered Number: 1072954 Registered office: Mortimer House, 37-41 Mortimer Street, London W1T 3JH, UK

Connection Science
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713411269

Learning Simple Arithmetic Procedures

Garrison W. Cottrell [a]; Fu-Sheng Tsung [a]
[a] Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA, USA

Online Publication Date: 01 January 1993

To cite this Article Cottrell, Garrison W. and Tsung, Fu-Sheng(1993)'Learning Simple Arithmetic Procedures',Connection Science,5:1,37 — 58
To link to this Article: DOI: 10.1080/09540099308915684
URL: http://dx.doi.org/10.1080/09540099308915684

PLEASE SCROLL DOWN FOR ARTICLE

Full terms and conditions of use: http://www.informaworld.com/terms-and-conditions-of-access.pdf

This article may be used for research, teaching and private study purposes. Any substantial or systematic reproduction, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The accuracy of any instructions, formulae and drug doses should be independently verified with primary sources. The publisher shall not be liable for any loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.

Learning Simple Arithmetic Procedures

GARRISON W. COTTRELL & FU-SHENG TSUNG

(Received for publication 16 October 1992; revised paper accepted 5 December 1992)

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

Rumelhart et al. *(1986b) proposed a model of how symbolic processing may be achieved by parallel distributed processing (PDP) networks. Their idea is tested by training two types of recurrent networks to learn to add two numbers of arbitrary lengths. This turned out to be a fruitful exercise. We demonstrate: (1) that networks can learn simple programming constructs such as sequences, conditional branches and* **while** *loops; (2) that by 'going sequential' in this manner, we are able to process* artibrarily *long problems; (3) a manipulation of the training environment, called* combined subset training *(CST), that was found to be necessary to acquire a large training set; (4) a power difference between simple recurrent networks and Jordan networks by providing a simple procedure that one can learn and the other cannot.*

KEYWORDS: Procedural learning, recurrent networks, cognitive modeling.

1. Introduction

A major criticism of artificial networks is that there is no obvious method for doing sequential, symbolic processing. Many authors assume this cannot be achieved by parallel distributed processing (PDP) networks, as noted by Fodor and Pylyshyn (1988):

> . . . by definition, connectionist architecture precludes the sorts of logico-syntactic capacities that are required to encode rules and the sorts of executive mechanisms that are required to apply them.

In a footnote to the above, Fodor and Pylyshyn note that "it *is* possible to simulate a 'rule explicit process' in a connectionist network by first implementing a classical architecture in the network". In the following, we show that one can train a connectionist network to follow rules without first going to the trouble of implementing, for example, a production system in the network, as some connectionists have done (Touretzky & Hinton, 1988). In addition, some have argued that symbol manipulation is simply "currently beyond the scope of the connectionist approach" (Hendler, 1989) and hence, in order to do such tasks, a hybrid system must be used. In this paper, we attempt to reduce the number of things currently beyond the scope of connectionism, rather than retreating to hybrids.

Rumelhart *et al.* (1986b) proposed that symbolic processing may be achieved

G. W. Cottrell & Fu-Sheng Tsung, Department of Computer Science and Engineering 0114, University of California at San Diego, La Jolla, CA 92093, USA. E-mail: gary@cs. ucsd. edu.

by (1) creating physical representations of the problem, (2) processing the representations vai 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, e.g.

```
327
865
---
```

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. Then, a carry must be written as well. Now a new pattern presents itself

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

and the process repeats. Similarly, Rumelhart *et al.* (1986b) claim that a complex logical problem is solved by breaking it down into simpler problems and apply the above procedure repeatedly. Doing a problem 'in your head' involves forming *internal representations* of the external representations used above. This latter aspect is beyond the scope of the current research. For our part, we were interested in just what was involved in implementing the above description.

There are a few problems with this description. First, no mechanism is given for how to perform a manipulation that might require several internal steps before the result is produced. Our model will use internal memory to allow several steps on a single input. Second, the pattern-recognition problem as stated above seems rather complex. We will finesse this by assuming that attentional processes can segment out the appropriate portion of the image, and that the image has been 'parsed' into an appropriate representation for the network. We will train the model to produce a signal when it is ready for the next portion of the input. Third, in this simple form, the proposal of Rumelhart *et al.* (1986b) is not very convincing as a model of sequential symbol processing—it is perhaps overly Skinnerian for modern tastes. We have chosen instead to model the task as one requiring a **while** loop with sequential processing of statements, internal variables and conditional branching.

We have chosen to use exactly this problem of addition to guide our foray into sequential symbolic processing by neural networks. We will show that it is possible to learn to add two arbitrarily large numbers with standard sequential networks. We chose addition as our symbolic processing task because it is a simple example that everyone is familiar with, yet it is non-trivial for PDP networks because it involves control issues as well. The procedure for addition involves sequential processing, conditional branching and looping, which are non-traditional PDP tasks. Furthermore, it is a good problem to test for generalization, because we obvously can only train the network on a finite set of examples while there are infinitely many new cases to test it on. The network must learn the implicit, underlying rule in addition, and it must exhibit systematicity in order to solve it.

As an unexpected benefit of this research, we discovered a method for training networks with large training environments which, at least in this domain, is

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

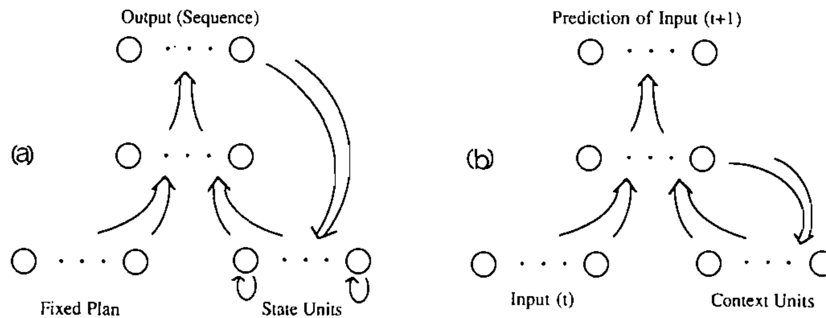


Figure 1. (a) Jordan's recurrent network. The input is a fixed plan, which determines which sequence the network will output. The outputs are linearly summed over time in the *state* vector, lower right, to keep track of the position in the sequence. (b) Elman's network. The network is trained to predict its next input. The context vector (lower right) is a copy of the hidden units from the previous time step to form a memory for the previous inputs.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

superior to giving the network a large subset of the environment initially. We found that our networks reached local minima when presented with large sets of examples. Instead, we developed an incremental method we call *combined subset training* (CST) that made the learning smooth and feasible. This method may be reasonable for other tasks, and may have implications for human learning.

The architecture of our model is agnostic between two familiar recurrent architectures—Jordan networks and simple recurrent networks (SRNs; see Figure 1). The former is usually used for sequence production, while the latter is used for sequence recognition. Our task is a blend of these two, and allows comparison between the architectures on the same task. Given the notion that the networks are learning a simple program, we are able easily to come up with a new program that one architecture can perform while the other cannot.

In the following sections, we discuss the architecture of the model, the training set and training regime in more detail, and the results on the addition task. We analyze the internal states of the model to show how it represents the programming constructs. Then we present a slightly modified version of the addition task that separates the two network architectures. Finally, we consider some implications of our model for PDP and cognitive modeling.

2. The Model

Our model assumes that the problem solver has the ability to focus attention on one column of digits at a time. There are then four actions that may be performed:

(1) *write(x)*, an action that records a result in the output. x is usually the low-order digit of the sum.
(2) *carry*, an action that simply states there is a carry, without writing it down.
(3) *next*, an action that shifts attention to the next column of digits. This changes what is presented as input on the next time step.
(4) *done*, an action that simply states the problem is finished.

Thus the input to the model at any given time is one column of digits. Several actions may be performed on this input before advancing to the next input. Also,

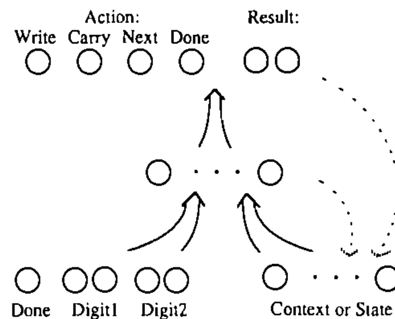


Figure 2. The network model of addition. The input is a representation of the two digits in the current column of numbers, along with a 'done' bit when the network has run off the end. The output is divided into two fields: an **action** field that represents the current action and a **result** field that represents the low-order digit of the sum when the action is WRITE.

the low-order digit of the sum depends on whether there is a carry or not on the previous sum. This fact is *not* in the input. Both of these considerations require the use of some form of memory.

In terms of the network model (see Figure 2), we used a binary representation of the two digits[1] as input, along with another bit which signals when the network has shifted beyond the last 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, state 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. In this case, the result field holds the lower-order digit of the sum of the two inputs. Otherwise, the result field is not important and no error is propagated back from it. The NEXT action gives the network control of when it is finished with this input and ready to move on to the next column of digits. An alternative would be to give the network a gray-scale version of the problem, and force it to recognise the digits and shift its attention internally. However, we wanted to keep the problem as simple as possible.

Given this set of actions, the network is trained to produce a sequence of outputs according to the following program:

PROGRAM 1

```
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, ???);
```

Notice that the CARRY action is performed only when there is a carry (else it is skipped). If there is a carry, there is no explicit representation in the next input telling the network that 1 should be added to the sum. That is, the network must

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

learn to 'remember' the carry. An example of a sequence of inputs and the corresponding outputs is given in Table I.

Table I. An example sequence of inputs and outputs (base 10). The circled numbers in the input column are the actual current inputs to the network. The horizontal lines mark the changes to the network input (all follow NEXT actions). Note that in step 6 the network must remember the carry in order to produce the correct answer

Input			Output: Action	Output: Result
3	2	⑦	1. Write	8
4	8	①	2. Next	—
3	②	7	3. Write	0
4	⑧	1	4. Carry	—
			5. Next	—
③	2	7	6. Write	8
④	8	1	7. Next	—
3	2	7	8. Done	—
4	8	1		

The features of this model are: numbers of arbitrary lengths can be added; conditional branching, looping, sequence-following and, of course, pattern association are all inherent in the process. The network must repeat the same series of actions on every input. This series of actions differs in number depending on conditions present in the input. The actual result of the computation depends on the processing history.

As we noted earlier, the model has to have a memory. 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. This can only be accomplished if the network has different states for different locations 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 (1990). Both of these approaches are restricted extensions of the basic feed-forward network used in most backpropagation experiments (Le Cun, 1985; Rumelhart *et al.*, 1986a; Werbos, 1974) that nevertheless still allow the use of backpropagation learning. 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. The state vector at time t becomes some proportion (μ) 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. 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

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

backpropagating one step in time along the recurrent links. Elman terms this architecture a simple recurrent network (SRN). We used both architectures in the following experiments.

3. The Training Set

We simplified the addition problem by using base 4 numbers instead of decimal numbers, with each base 4 digit represented in binary. The use of base 4 vs base 10 lessens the burden on the network of the number of basic pattern associations it must memorize. There are now 16 basic associations (4×4, although they are symmetric), plus special conditions such as the presence of a carry. The carry actually complicates the situation quite a bit; an extreme way of counting the associations the network has to learn would be to count 16 possible previous inputs (which determine whether or not a carry is added into the sum) times 16 possible current inputs for 256 associations. This is only counting the WRITE steps. An even more extreme way of counting the WRITE steps would be to recognize that the network has essentially an infinite set of different inputs depending on its processing history. This is reflected in the context or state vectors. Since most of this is irrelevant, one thing the network must learn is to *ignore* its distant history.

The goal is to train the network to perform addition on an arbitrary number of digits. Immediate questions are: What makes a good training set? How many examples are enough for the net to generalize? We decided to restrict our attention (during training) to additions with up to three digits. This set contains all the canonical situations, and therefore should be enough to get the basic idea across to the network. After this initial phase, we use additional training to 'clean up' the network's processing on longer additions.

4. Training Procedure

Training turns out to be a more difficult problem than it first appears to be. To see why, consider that there are 4096 pairs of one- to three-digit base 4 numbers. When these are expanded into the network input–output sequences, there are more than 30 000 individual patterns. Learning is very difficult with such a large training set. To keep the network small (16 hidden units), and the training fast (less than a minute per epoch on a Sun-4), we tried two training environments. One was a random subset consisting of 1% of the 4096 additions. This was learned quickly, but generalization was poor. We found if we tried a bigger subset of 8% of the additions, the network would hit a local minimum. 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 ensure good generalization, the network does not learn in the time we are willing to wait.

It seemed that another method of training was needed.

4.1. Combined Subset Training

Even though the training set is very large, there is a great deal of structure in it. We decided to try to take advantage of this structure by training the network on

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

a small subset of the task, and then increasing the training subset size until it generalized to the whole training set. We call our technique the *combined subset training* (CST) method. The idea is very simple: we pick a manageable, randomly selected subset to train the network initially. When the network has learned this subset fairly well, we add another randomly selected subset of equal size to the first training set, and then train the network with the combined set. The network is now being trained with a set that is twice as large, but it already 'knows' half of them. If this can be learned successfully, we double the training set in the same fashion. This procedure is repeated until the whole set is included, or until we find that the network is able to generalize to the rest of the original training set. This is a random selection method that grows the training set during learning (cf. Elman, 1991b; Plutowski *et al.*, 1993).

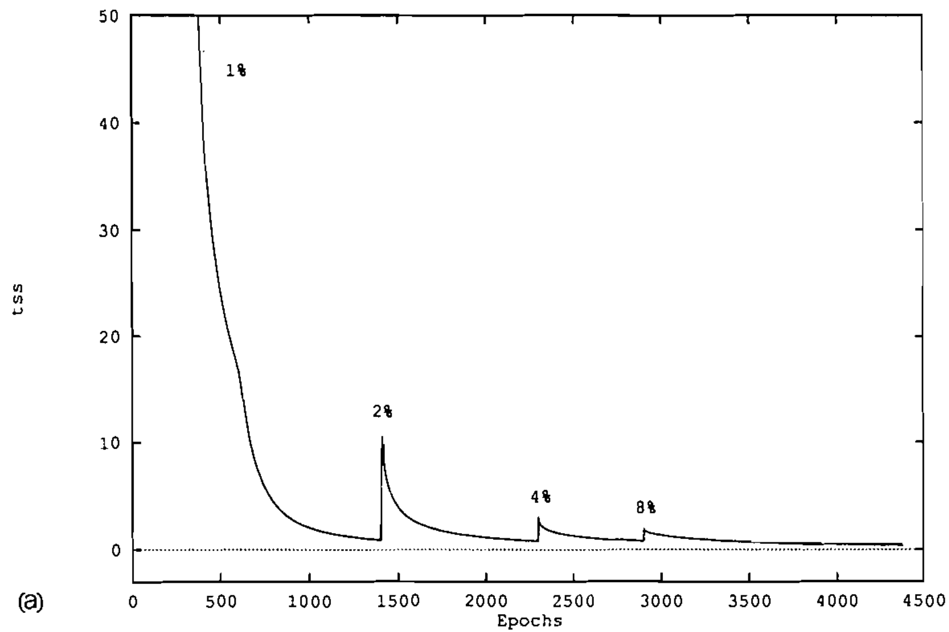
In order to apply this procedure, one has to find a manageable subset to begin with that the network can learn easily. In our case, 1% of the training set was learned quickly, so we started with that. Our procedure for deciding when to double the subset was to do so when the error curve was 'flat' over a sufficient number of epochs (e.g. 2000), or if the total sum-squared error (tss) was half of the tss prior to the previous doubling over a smaller window (e.g. 500 epochs). In what follows, we decided 'flatness' by eyeball, but automating this is trivial. For example, one can use the 'trigger slope' idea of Ash's (1989) dynamic node creation procedure. We stop when the network generalizes to the rest of the training set. Throughout this procedure we use a single learning rate of 0.02.

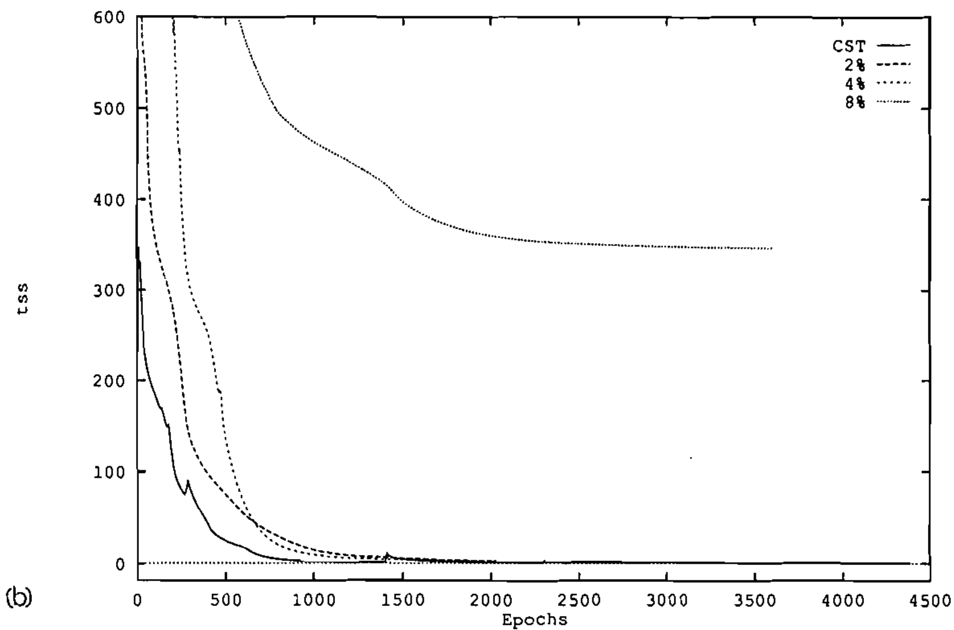
For this initial experiment, we picked a random 1% of the 4096 three-digit additions, along with the 16 one-digit sums as our initial test. We trained the networks to a tss of 1.0 and doubled the training subset. The tss for the doubled training set initially jumps up, but monotonically decreases quickly (see Figure 3(a)). The same behavior was observed when we doubled to 4% and 8%. Thus each jump in the error graph corresponds to a doubling of the training set size. It is characteristic of a successful training regime that the peaks of these jumps are decreasing. At 8% of the training set (328 addition examples), we found that the network generalizes very well to the rest of the 4096 additions.

In Figure 3(b), we compare CST with training directly on 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 minimum at tss 346, while the networks trained on 2% and 4% do not generalize well to the rest of the patterns (data not shown). This is evidence of the dilemma we stated above: when training on *fixed* subsets, if we keep the set size small enough for the network to learn, it doesn't generalize well. If we start with bigger subsets, the network doesn't learn. Also, the 8% curve is as smooth as it is because of manual intervention to reduce the learning rate when the error began to oscillate. Figure 3(c) shows two 8% networks, one where we lowered the learning rate when oscillations began, and one where we didn't. In this case, the network eventually smooths out anyway, but this is not always the case. The point is, annealing the learning rate did not work, although we obviously did not try all possible schemes.

Generalization to the rest of the training set is determined by dividing the entire 4000 additions randomly into 10 files, each containing 10% of the total training set, and testing the network with all 10 files. This allows us to see if the error is spread evenly over the entire set. In each case, the tss (over an individual file of about 400 additions) was between 0.4 and 0.7. That is, the average error for each addition (which may include up to 11 steps) was below 0.002. More importantly,

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008





this suggests that the error was spread very evenly among the sequences. There can be no case in which the network outputs a 1 when it should have output a 0 (or vice versa), because that immediately adds about 1.0 to the tss. The results are similar for both the Jordan network and the SRN.

To test generalization of the CST-trained network to longer additions, we next

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

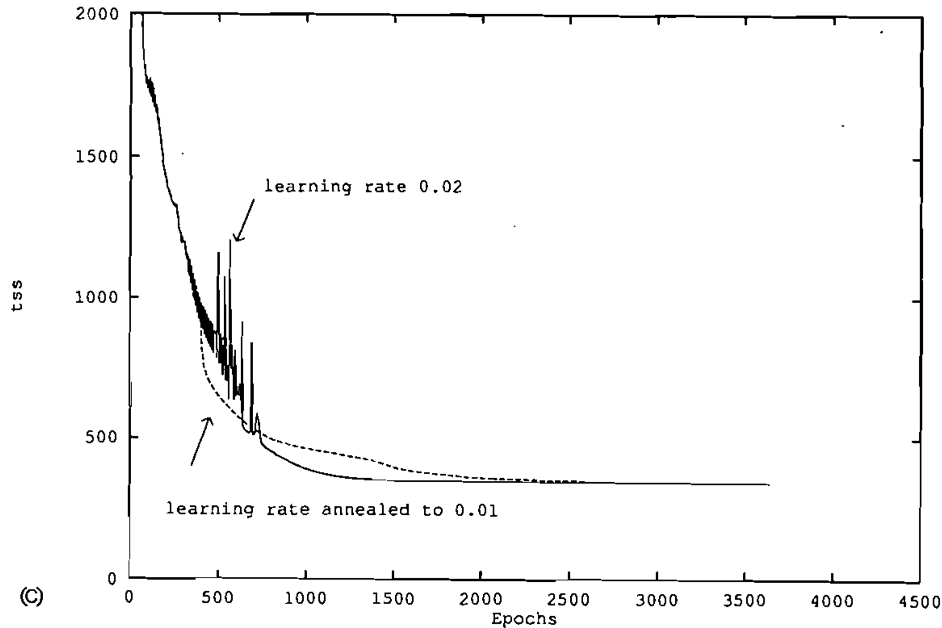


Figure 3. (a) Error curve from combined subset training with the SRN. Each jump in tss is where training sets are doubled. Beginning tss: 348.5; final tss: 0.399. (The error curve for the Jordan net is similar.) (b) Comparison of CST to fixed set training for 2%, 4% and 8% subsets (note the y-axis is much higher than in (a)). (c) Two separate runs of the 8% training set (no CST). The smoother curve was obtained by lowering the learning rate (from 0.02 to 0.01) during training.

generated arbitrary base 4 additions with numbers of lengths up to 14 digits (8 decimal digits). We generated 10 test sets containing 10 such additions each. An example test set is as follows:

78965	+	87	=	79052
9432080	+	92	=	9432172
40	+	76084	=	76124
7	+	8	=	15
94112	+	9133431	=	9227543
5078	+	83244	=	88322
94	+	9239650	=	9239744
30163504	+	6	=	30163510
74157	+	159	=	74316
389775	+	855	=	390630

Even though the network has only been trained on additions of up to three digits, on the 10 test sets the network missed only 1 addition out of 10 on average. This data is shown in the first column of Table II. We then performed a 'clean up' phase during which we trained the network on the test set with the maximum error for 10 epochs at a time until the error was acceptable on all test sets. This procedure resulted in 30 epochs on test set 1, followed by 10 epochs on test set

9 (cf. Plutowski *et al.*, 1993). The tss errors for all files, after every 10 epochs, are shown in Table II. We then tested the new weights on both the previous three-digit additions and hundreds of the arbitrary, large additions. The errors for the latter are shown in Table III. In general, we find that:

(1) Training on one or two test sets corrects performance on the other test sets. This suggests that there is only one or a few types of error involved.
(2) The network learns to correct its mistakes quickly. Furthermore, the new weights do not upset the performance on the previous additions.
(3) The refined weights then generalize to arbitrary problems with errors of less than 0.5%.

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.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

Table II. Generalization behavior and weight refinement from mistakes. Each row corresponds to one test set of 10 arbitrary additions. The first column is the total sum-squared error (tss) after the network has learned only on three-digit additions. The weights are then refined by training on test set 1 for 30 epochs, which raised the tss for test set 9. The last 10 epochs are therefore trained on test set 9

Test set	Initial tss	Test 1 10 epochs	Test 1 20 epochs	Test 1 30 epochs	Test 9 10 epochs
1	24.229	19.107	1.493	0.389	0.215
2	0.147	0.166	0.083	0.101	0.146
3	0.198	0.143	0.108	0.129	0.178
4	14.152	6.810	0.127	0.147	0.221
5	18.063	5.192	0.109	0.133	0.333
6	14.675	2.687	0.539	0.212	0.154
7	10.100	5.776	0.125	0.115	0.252
8	0.905	0.393	0.072	0.070	0.088
9	6.693	0.509	0.378	5.684	0.344
10	8.347	2.014	0.599	0.637	0.204

The fact that training on one set reduces the errors on others suggests that the network is making just one type of error. By inspection, we determined that the error occurs when the network is adding a small number to a large number. After the end of the smaller number, the network is repeatedly adding 0 to a long string of digits. At the output level, one can observe the activation on the 'wrong' action creeping up over time, until the network eventually outputs the incorrect action. At that point, the network ouputs are out of phase with the correct outputs, which produces high errors for the rest of the addition. This is due to the fuzzy internal representation of the state of processing as a region of state space (see the next section). The network has not been trained to maintain these distinctions over a very long sequence, and eventually the internal representation drifts from the correct state into the incorrect region of state space. The extra training sharpens up this representation, preventing drift.

We only found one trial (among several successful trials) where the CST method failed. This was on a somewhat harder task (see later). The reason for the failure is probably an unfortunate set of initial weights, since the other runs on this

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

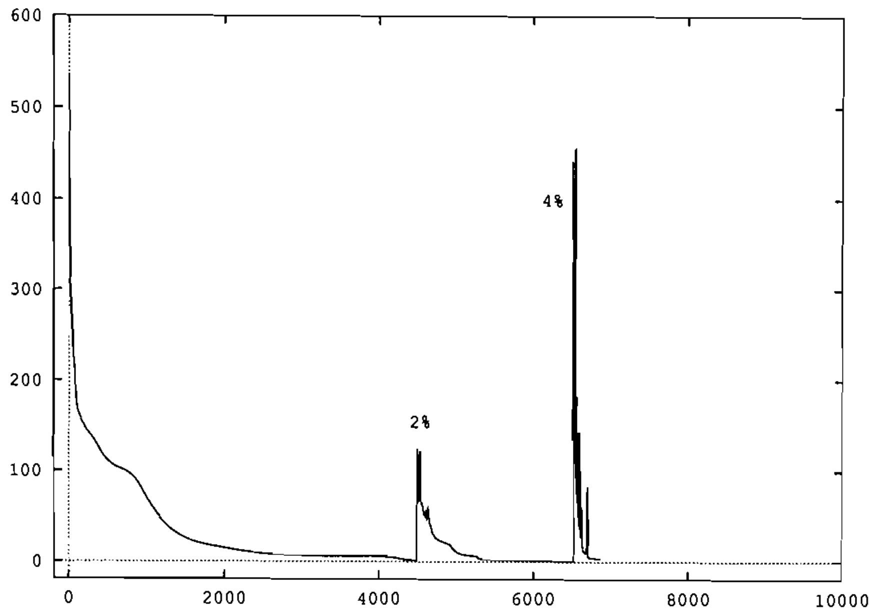


Figure 4. An example of an unsuccessful CST run on a harder problem. Note the error peaks increase. This is a signal to abort this run.

Table III. Total sum-squared errors from 10 more test sets, each containing 50 additions. Refinement refers to the 40-epoch training in Table II; no further learning is done on these tests. Refinement improved performance drastically for arbitrary additions, while degrading very little on random 2% and 4% subsets from the original three-digit examples

Test sets	Before refinement	After refinement
11	60.328	0.807
12	69.133	0.688
13	84.252	0.637
14	42.185	1.351
15	51.162	1.054
16	108.05	0.909
17	80.287	0.864
18	35.224	0.858
19	56.477	1.273
20	97.135	0.682
2%	0.119	0.455
4%	0.216	0.882

problem converged. Figure 4 shows the behavior of the error curve on this harder problem for the network that failed. The important point here is that the CST method provides a simple diagnostic as to whether to continue training or to restart. The error peaks when the training set size is doubled are increasing rather

than decreasing. Using this diagnostic aspect of CST to decide to restart can mean great savings in computation time for large runs.

5. The Model's Representation of Programming Constructs

Given that the network has learned the programming constructs, how are these represented by the network? One can apply standard cluster analysis techniques to the hidden unit representations to view the similarity structure the network imposes upon its inputs. However, these are static representations of a dynamic process. It is more interesting in this case to look at the *dynamics* of the network as it moves through the problem. To look at this, we found the principal components of the 16 hidden unit activations over time, as an SRN processed 10 additions from one of the generalization test sets from Table 2. This 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. This is the same kind of analysis that is used in Elman (1991a).

We show two kinds of plots from this analysis. In Figure 5, we show the projection of the hidden unit vectors on to 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 is a projection of the phase portrait of the internal states of the network as it processes the input. There are several interesting points to make about this graph. The **while** loop is represented as the triangular trajectory in this graph. The vertices of these triangles represent the different actions that the network can perform in the **while** loop. Thus the network divides up the space according to the current action being performed. WRITE result actions (labeled R#) are generally in the right half of the space, while NEXTs and CARRYs are in the left. (Differences required to generate different outputs in the digit field that are not exhibited show up on other principal components.) The 'true' condition on the conditional statement is represented by the trajectories that proceed from WRITE result actions to CARRYs versus the 'collapsed' trajectories directly between WRITE result and NEXTs. Also, note that with a few exceptions, the second component is correlated with the current step of the problem. It is interesting that the network represents this even though it doesn't need to in order to solve the problem. Finally, we can see that along 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. It is 'remembering' this fact in how it internally represents the state following a carry. This internal representation is available at the next iteration as the context vector. Hence it can solve the problem. Note that this difference in internal state is not reflected at the output level at this point since in both cases, the network output is 'NEXT ???'.

Another way of plotting the movement of the system through its internal state

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

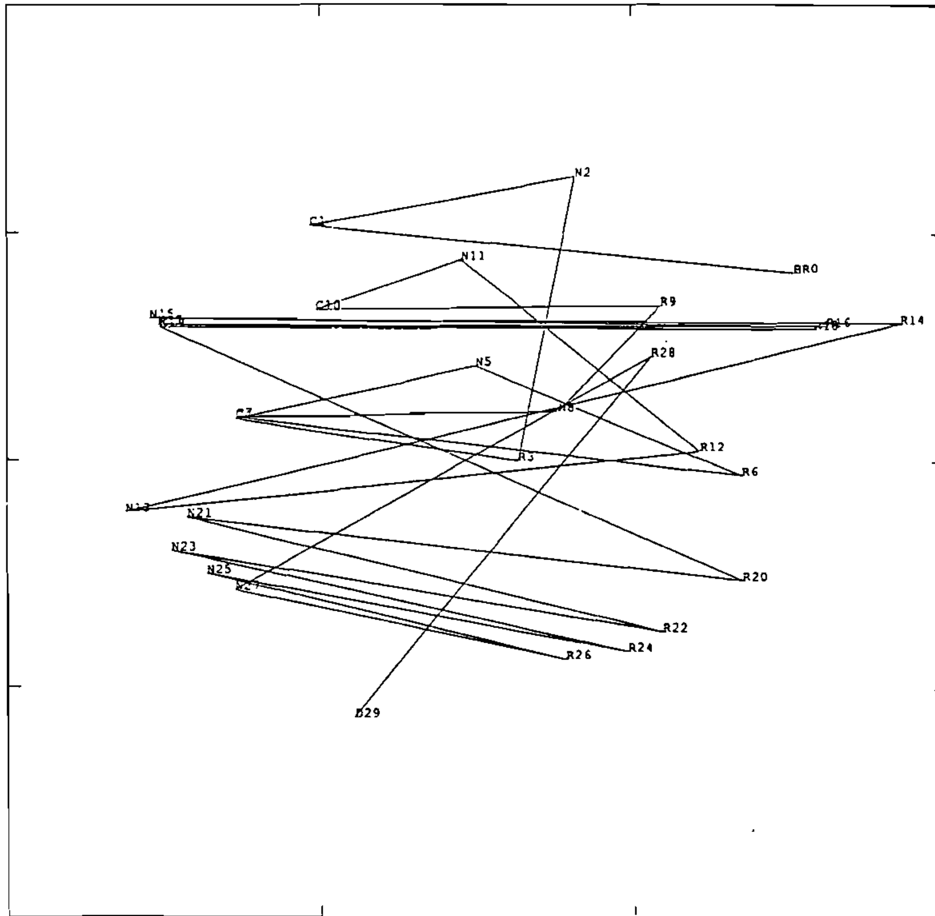


Figure 5. Projections of the hidden unit activation vector on to the first two principal components as the network is carrying out a 30-step addition. The number in the point labels corresponds to the step in this addiction. R: WRITE (RESULT), C: CARRY, N: NEXT, D: DONE. The vertical direction is P.C. 2.

space is to plot the first principal component at time t vs $t + 1$. This essentially gives the map from the context vector to the next hidden unit vector, or the 'next-state' mapping of this automaton, since the context vector at time t is the hidden unit vector at time $t - 1$. In Figure 6, we show this plot for the same problem as in Figure 5. Here points are labeled by the transition being made. So N5→R6 means we are plotting the point (hidden(5), hidden(6)), i.e. the hidden units at time steps 5 and 6, and the output in each case is 'Next ???' and 'Write ##' (again, 'R' stands for 'write Result'). Here the separation of the NEXTs following a CARRY becomes obvious—they are all in a 'bump' above the main diagonal of the graph. This is another view of the corner of the internal state space that represents memory of the carry bit. Note that for the SRN, this information is not explicitly represented in its state as it would be for a Jordan network working on the same problem. Thus this kind of graph allows us to view directly the internal dynamics of the network as it is processing the input, revealing 'hidden' variables there.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

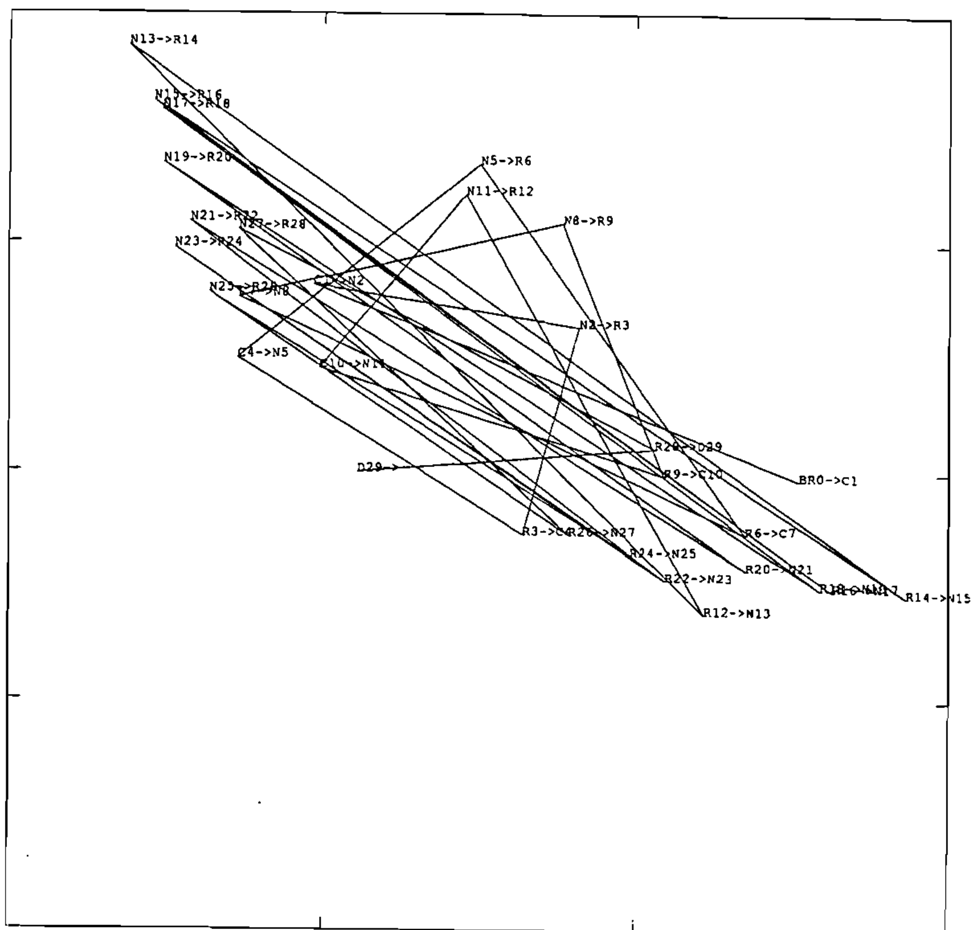


Figure 6. Projection of the hidden unit activation vector on to principal component 1 plotted against itself one time step later. This is from the same sequence plotted in Figure 5.

6. Differences between Jordan Networks and SRNs

Both Jordan networks and SRNs behaved similarly on this project. We may ask the question: Is there some task that one can do, and not the other? Do they have important differences? It is a feature of treating the network as executing a program that we can construct a simple example by merely switching the order of a pair of steps in the program the network must learn. That is, instead of the original sequence of 'write result-carry-next', the network is trained to output 'write result-next-carry', as in the following:

PROGRAM 2

```
while not done do
begin
  output(WRITE, low_order_digit);
  output(NEXT, ???);
```

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

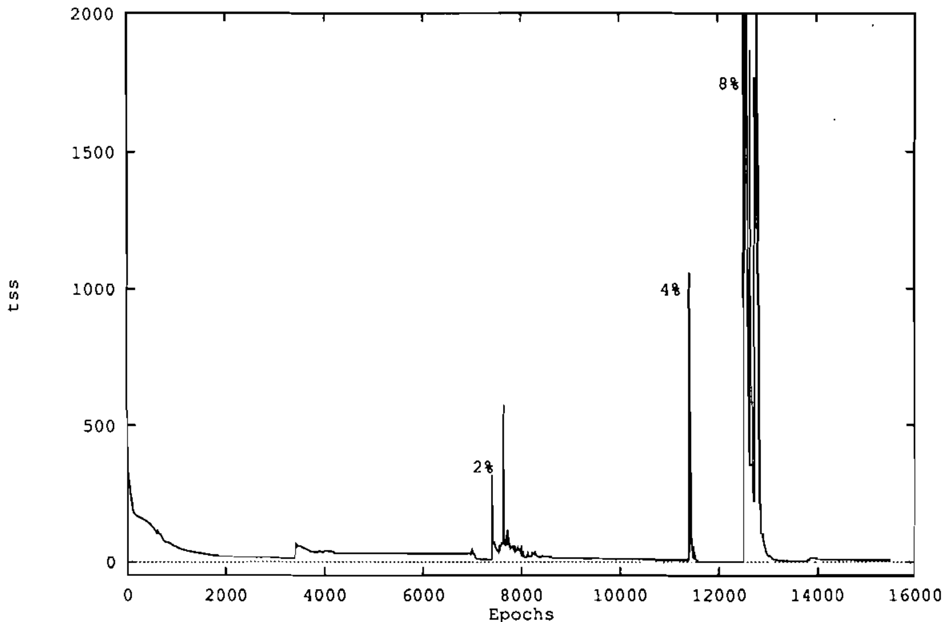


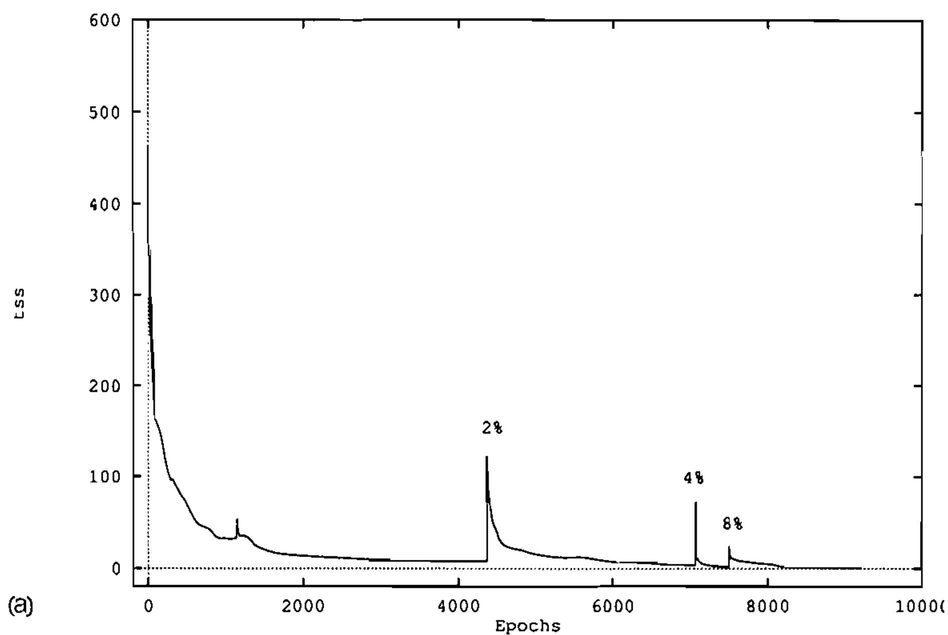
Figure 7. Learning Program 2 with the Jordan network. CST doubling is indicated. The network is memorizing sequences, and is unable to generalize.

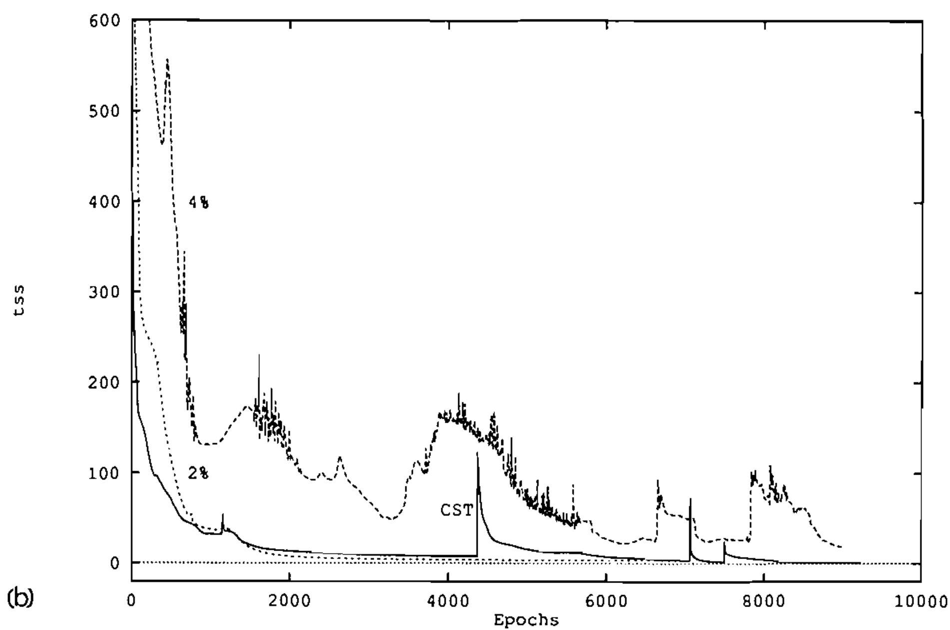
Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

```
if sum(previous_input) > radix then
  output(CARRY, ???);
end
if carry_on_previous_input then
    output(WRITE, '01');
output(DONE, ???);
```

The Jordan network should not be able to solve this problem. This is because the Jordan 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.[2] After writing the result, the NEXT action brings in the next pair of digits as input, so that the Jordan net cannot possibly determine whether the next step should be CARRY or not. The SRN has a transformed version of the input at the hidden layer which is recycled at each time step, thus it can 'remember' input that is not reflected in its output.

Simulation results bear this out. The Jordan network has a problem learning this task, as shown in Figure 7. It can achieve 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 SRN show that this is a harder problem than the original problem. The graphs in Figure 8 compare CST to the fixed subset methods on this problem. Since this is a harder problem, this task takes CST twice as long to learn as the previous task did.





7. Discussion

7.1. Learning Programming Constructs

This project demonstrates that PDP nets can do simple sequencing, looping and branching. It should be possible to construct reasoning systems that use this kind

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

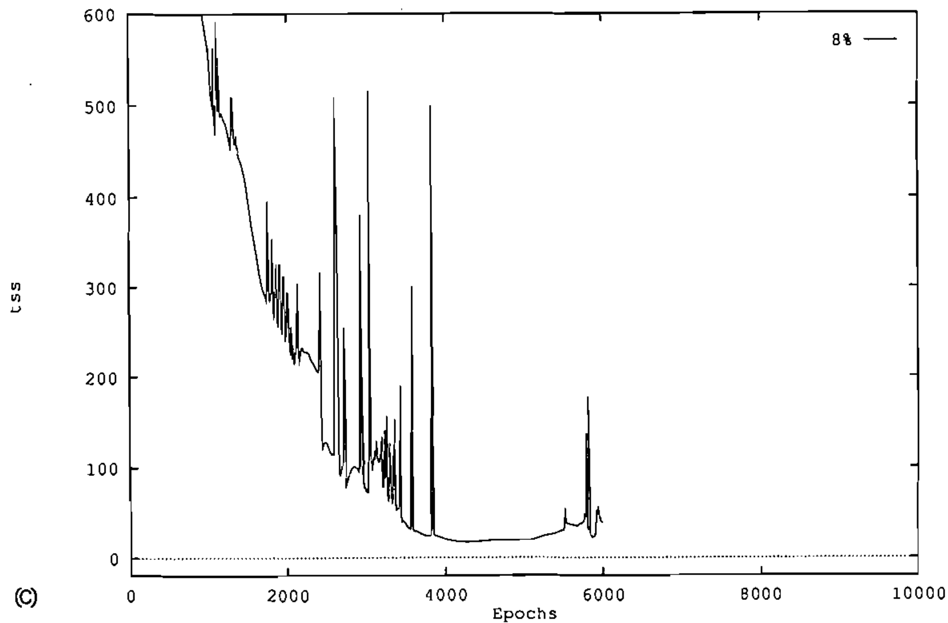


Figure 8. (a) Error curve of learning Program 2 with the simple recurrent network. Using CST up to 8% is sufficient for the network to generalize. (b) CST vs 2% and 4% fixed set training on the same task. (c) 8% fixed set training on the same task.

of architecture, by training the 'rules' into the input–output behavior of the network. However, we have not demonstrated that **while** loops can be nested. We conjecture that they can using this architecture, but that there will be strict limits on nesting depth.

Also, we showed that values necessary for future processing can be stored over short periods by the simple recurrent network. Other recurrent network models are more powerful in this regard (William & Zipser, 1989), but require a prohibitively large amount of computer time to train. However, since remembering even a single bit takes a long time to learn, it suggests that memory for 'variables' requires initial structures subserving this function that are easily refined by learning. One may surmise that this is what evolution does for us, and we should not try to learn the memory function. We return to this point later.

Given a memory mechanism, we thus have simple versions of all of the mechanisms for a universal computer, except the ability to nest these constructs to infinite depth. We have conjectured that nesting will not be able to be carried very deeply. Thus it would be very interesting to set up a problem with nested **while** loops.

It is important to point out that, as far as we know, this is the only connectionist network that can do arbitrarily long addition problems. This ability lies in being able to 'go sequential', that is, dividing the problem into small chunks that can be solved relatively independently, one after the other. Obviously, we trained the network explicitly to do this. It poses a challenge to future researchers to come up with methods for a network to divide a sequential task into chunks on its own.

7.2. Combined Subset Training

We suggest that the *combined subset training* (CST) method for large training sets is superior to the 'all-at-once' method usually employed. Of course, application of this method to other problems is required to check out this claim. We applied it to a problem where the training set is large and unwiedly, yet the problem space is fairly 'uniform'. Experience with CST on this project shows that one can keep the same learning rate throughout the training, even when the training set is doubled. Also, the number of epochs necessary to reach error criterion after doubling the training set size is often *smaller* than the number needed for the previous subset. The total error is usually bounded by the total error of the initial subset, and in fact each doubling raises total error only slightly (if the network is converging to the right solution). The main point, of course, is that CST allows us to reach the criterion on extremely large training sets.

7.3. A Power Difference between Jordan Networks and SRNs

This work uncovered a simple difference in the computational power of Jordan networks and Elman's SRNs. Basically, Jordan networks cannot remember things about their input that are not reflected on their output.[3] Does this mean that Jordan networks are strictly less powerful? The answer is no, and it is also simple to see why this should be. The 'next-state' computation of the Jordan network is from the current state and the input, through the hidden layer, to the next-state in the output (plus a simple linear operation). The 'next-state' computation in an SRN is from the context vector and the input to the hidden layer.[4] Thus the SRN can only compute *perceptron-reachable* next-states, whereas the Jordan network can do more. For example, the Jordan network's next-state can involve an exclusive-OR computation. The catch is that the next-state is determined by the teaching signal, not the needs of the problem. This suggests that an SRN with an extra hidden layer between the context and the original hidden layer would be the best of both worlds. Allen's (1990) architecture looks like this, but the weights from the previous hidden activations are not trained.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

7.4. Cognitive Predictions

The model makes several predictions about procedural learning. First, learning will work best by starting with small sets of examples and increasing the numbers of examples until the entire concept is learned. The work reported here suggests that there are reasons other than attention span why this might be a good idea. Specifically, learning a small number of special cases allows the learning process to reduce the dimensionality of the search process by using a few examples to narrow the space.

Second, other partitions of the training space may speed up learning. We happened to double the size of the training set each time, and found that the error curves consistently fell under the errror curves for all-at-once training. We may be able to achieve more speed-up with different rates of increasing the set size. This would then be a testable process with children.

Third, the result that it takes much longer to learn to compute and remember the carry bit when the source of the carry bit is not available (the change from Program 1 to Program 2) suggests that *it is easier to remember something if you're*

looking at it! This is a bit facetious perhaps, but just as in humans, when a task requires a memory load, it takes longer to learn. The task in Program 2 requires the network to 'remember' input across one extra processing cycle, over the task in Program 1. This second task takes almost twice as many epochs to learn.

This point is worth taking a bit farther. Note that in order to carve this *one bit* memory out of its internal state space, the SRN took *5000 extra epochs.* This strongly suggests that learning is not the right way to go about solving the memory problem. Rather, nature has most likely provided us with ready-made structures for remembering things. This suggests the use of different PDP models that do fast associations, such as the CHARM model (Metcalfe, 1983, 1985, 1991, in press), in conjunction with SRNs. Another alternative is to build in memory structures that can be tuned as in Jain (1989).

Some might argue that this network does not learn at all the way children do. Children are given an explicit rule to follow for addition. The basic issue is learning by personal experience versus learning by being told. It is certainly still an open question whether there are ways in which connectionist networks can learn facts and procedures by being explicitly given them. However, we feel that this aspect of learning is overemphasized and overrated. When I am given a verbal description of how to make coffee using the new coffee machine, it is often not enough to carry out the procedure perfectly, and I may forget how to do it the next time if enough time passes. The procedure is not really learned until *I have carried it out myself several times.* This fact is reflected in the common experience that students do not really learn concepts well without homework problems on them.

The explanation of how someone learns something by being told rests in the somewhat metaphorical idea of pushing the network into a different region of activation space.[5] For example, the Necker cube network (Feldman & Ballard, 1982), essentially learns by 'being told' which way to view the cube. The inputs push it into a region of state space that corresponds to that interpretation of the input. St John and McClelland's (1990) sentence-processing network is a higher-level version of this same idea. Once the network's state space has been shaped by learning, it is then ready to be told a proposition (in the form of a sentence) which it then stores in its internal state. It can then 'answer questions' about this sentence, such as who did what to whom. In this sense, its behavior has changed based on what it was told. A complete model would then deepen the attractor well of this pattern in order to store it.

8. Future Work

There is certainly some more work to be done. For example, we would like to analyze the network representations developmentally, as it learns, and observe the bifurcation of the internal representation that is necessary to solve the problem. Perhaps by looking at the state space of the networks we can get a better idea of why CST works better than fixed subset training. Another issue is whether the subspace that corresponds to doing the procedure forms an attractor. If we start the network in an abnormal state, is it drawn to the space that corresponds to solving the problem?

Also, it would be a challenge to get a network to build internal representations once it has learned the external process. That is, doing the problem 'in the

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

network's head'. Simulating this process would require scanning the inputs (simulating receiving a verbal description of the problem), remembering them and doing the problem. This would require shifting attention on internal representations, rather than the simple external shifts implemented here. It would also require larger memory capacity in terms of short-term memory than that demonstrated here. Intermediate results need to be kept and combined for giving the final answer. This is difficult enough for humans, and seems an extremely challenging task for a neural network.

A more obvious direction to take this work is subtraction. Subtraction is a much harder task, involving nested **while** loops when borrowing across zeros. Preliminary work has been done (Nguyen, 1989) that shows the networks can perform borrowing from the neighboring digit, but so far we have not pushed this to the borrowing across zero stage.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

9. Conclusions

In this paper we presented a simple model of symbolic manipulation using a connectionist network that learned a procedure to add two multi-digit numbers. The model is a demonstration that networks of this type can learn simple programming constructs that are not nested. It is conjectured that the ability to nest constructs will be severely limited. Related evidence can be found in work by Cleeremans *et al.* (1989) on learning regular languages with simple recurrent networks. They found that nesting one regular language inside another caused the network to 'forget' where it was in the outer language.[6] More powerful networks are needed to learn such languages (Zipser, personal communication).

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 CST. 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.

A 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 and 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.

Although we did not set out to build a cognitive model of learning addition, we believe that our model may have some important implications for learning by children. It suggests that it is best to learn by training to a reasonable performance on a small number of examples, even if they may have some idiosyncrasies, then adding more examples while retaining the old ones. 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. We find regions of the internal state space correspond to states of the finite-state control necessary to carry out the problem. This type of analysis is necessary when we are using recurrent networks to observe the dynamics of the system.

Acknowledgements

This work benefited greatly from the comments of Jeff Elman, an anonymous reviewer, and members of the AI, GURU and PDPNLP Research Groups at UCSD. This work was supported in part by NIMH grant 1 RO3 MH46054-0.

Notes

1. A thermometer or 'dipstick' encoding would have made the problem easier.
2. Note that altering the μ parameter has no effect on this argument. It is based on a simple logical property of the network's memory.
3. One might expect that some extra outputs not trained to throw away input information may help this situation. Preliminary experiments with randomly wired extra outputs in a Jordan network did not produce an ability to remember the inputs.
4. Note that the 'next-state' in one is computed using the *output* of the network, and in the other it is computed from the *hidden layer* of the network.
5. We are indebted to Paul Churchland for pointing this out.
6. Biasing the probabilities on the arcs within two instances of the sublanguage helped the network to learn, but not to perfection.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008

References

Allen, R.B. (1990) Connectionist language users. *Connection Science,* **2,** 279–311.

Ash, T. (1989) Dynamic node creation. *Connection Science,* **1,** 365–375.

Cleeremans, A., Servan-Schreiber, D. & McClelland, J.L. (1989) Finite state automata and simple recurrent networks. *Neural Computation,* **1,** 372–381.

Elman, J. (1990) Finding structure in time. *Cognitive Science,* **14,** 179–211.

Elman, J. (1991a) Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning,* **7,** 195–225.

Elman, J. (1991b) Incremental learning, or the importance of starting small. In *Proceedings of the Thirteenth Annual Conference of the Cogtnitive Science Society.* Hillsdale, NJ: Erlbaum.

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

Fodor, Jerry A. & Pylyshyn, Zenon W. (1988) Connectionism and cognitive architecture: a critical analysis. *Cognition,* **28,** 3–71.

Hendler, J. (1989) Editorial: on the need for hybrid systems. *Connection Science,* **1,** 227–229.

Jain, A.N. (1989) *A Connectionist Architecture for Sequential Symbolic Domains.* Technical Report CMU-CS-89-187, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Jordan, M. (1986) *Serial Order: A Parallel Distributed Processing Approach.* Technical Report 8604, Institute for Cognitive Science, University of California, San Diego, La Jolla, CA.

Le Cun, Y. (1985) Une procédure d'apprentissage pour réseau a seuil assymétrique [A learning procedure for asymmetric threshold network]. *Proceedings of Cognitiva 85,* pp. 599–604. Paris.

Metcalfe, Eich, J. (1982) A composite holographic associative recall model. *Psychological Review,* **89,** 627–661.

Metcalfe, Eich, J. (1985) Levels of processing, encoding specificity, elaboration, and CHARM. *Psychological Review,* **91,** 1–38.

Metcalfe, J. (1991) Recognition failure and the composite memory trace in CHARM. *Psychological Review,* **98,** 529–553.

Metcalfe, J. (1993) Novelty monitoring, metacognition, and control in a composite holographic associative recall model: implications for Korsakoff amnesia. *Psychological Review,* **99,** in press.

Nguyen, Mai (1989) *A Cognitive Model of Learning Subtraction.* Internal working paper, CSE Department, UCSD.

Parker, D.B. (1985) *Learning Logic.* Technical Report 47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA.

Plutowski, M.E., Cottrell, G.W. & White H. (1993) Learning Mackey Glass from 25 examples, plus or minus 2. *Advances in Neural Information Processing Systems 5.* San Mateo, CA: Morgan Kaufmann.

Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986a) Learning representations by backpropagating errors. *Nature,* **323,** 533–536.

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

St. John, M.F. & McClelland, J.L. (1990) Learning and applying contextual constraints in sentence comprehension. *Artificial Intelligence*, **46**, 217–257.

Touretzky, D. & Hinton, G. (1988) A distributed connectionist production system. *Cognitive Science*, **12**, 423–436.

Werbos, P. (1974) *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Unpublished Doctoral dissertation, Harvard University, Cambridge, MA.

Williams, R. & Zipser, D. (1989) A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, **1**, 270–280.

Zipser, D. (1989) Personal communication.

Downloaded By: [CDL Journals Account] At: 22:38 30 October 2008