Evaluation of Two Connectionist Approaches to Stack Representation

Evaluation of Two Connectionist Approaches to Stack Representation

Rebecca Hwa

TR-08-97

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Evaluation of Two Connectionist Approaches to Stack Representation

Rebecca Hwa

Abstract

This study empirically compares two distributed connectionist learning models trained to represent an arbitrarily deep stack. One is Pollack's Recursive Auto-Associative Memory, a recurrent back propagating neural network that uses a hidden intermediate representation. The other is the Exponential Decay Model, a novel architecture that we propose here, which tries to learn an explicit represention that models the stack as an exponentially decaying entity. We show that although the concept of a stack is learnable for both approaches, neither model is able to deliver the arbitrary depth attribute. Ultimately, both suffer from the rapid rate of error propagation inherent in their recursive structures.

1 Introduction

A stack is one of the most fundamental abstract data-structures in Computer Science; it forms the bedrock of traditional symbolic systems. And yet, when we move into a distributed connectionist framework, finding a good representation for such a basic data-structure is surprisingly difficult. One could, of course, hand code a neural network to mimic the behavior of a stack under the symbolic system (e.g., some form of a fixed sized shift register), but that would circumvent the advantages of connectionism, such as the ability to learn and generalize and process noisy data.

Can a neural network learn to represent a stack from just seeing training examples? This question has been addressed by connectionist researchers, but many of the resulting stacks are so monstrously complicated that they cannot be practically integrated into other networks as their underlying data-structure. One exception is Pollack's Recursive Auto-Associative Memory (RAAM) [11], whose relatively simple architecture has allowed other

connectionist applications to use it as their stack components [1] [9]. Using a recurrent network architecture, RAAM attempts to simulate a stack of arbitrary depth, whose current state is encoded in a hidden intermediate representation. In this paper, we empirically determine that RAAM's ability to model a stack is limited to a shallow stack of depth 10. We propose a new stack representation, the Exponential Decay Model. Like RAAM, the Exponential Decay Model uses the recurrent network architecture to simulate a stack of arbitrary depth, but instead of viewing the stack state as a black box, the Exponential Decay Model makes the intermediate representation explicit. We present the results of empirical comparisons of the two representations. While the Exponential Decay Model does learn the specified stack representation, it does not scale up as well as RAAM so that the stack it represents is even shallower. Ultimately, both approaches suffer from the rapid rate of error propagation inherent in their recursive structures.

The rest of the paper is organized as follows. Section 2 describes the network architectures of the two stack models and the learning functions on which they are trained; next, in section 3 we define the evaluation metrics; in section 4 we analyze the effects of different training strategies and sizes of the network on the performances of the models; finally, we interpret the results of the experiments in section 5.

2 Stack Representations

Although both RAAM and the Exponential Decay Model are constructed as recurrent networks, their different philosophy on stack state representation drives them to very different learning functions. We first see how RAAM, by learning the identity function, can be interpreted as simulating a stack. Next, we show how a stack can be thought of as a set of floating point values under the Exponential Decay Model.

2.1 Recursive Auto-Associative Memory

RAAM is a standard 3 layered M-N-M network, where M is the number of input/output units, N is the number of hidden units, and $N \leq M$. The network learns the identity function from training examples where the input is the same as the target output. Pollack shows that if the identity function is perfectly learned and if recurrence is allowed, it can simulate a stack of arbitrary depth. The input layer of the network is made up by a concatenation of a representation of both the symbol to be pushed and the

current stack state. If the system receives a push command, the new symbol and the current stack state is compressed into a new stack state, which is represented by the hidden layer. To pop a stack, the system decompresses the hidden layer by feeding it as input to the second half of the network to extract the top symbol of the stack and the stack state after the pop, stored in the output layer. If the system receives two consecutive push commands, the pattern stored in the hidden layer is copied into the stack state in the input layer after the first push. Similarly, if the system receives two consecutive pop commands, the pattern stored in the stack state of the output is copied back into the hidden layer after the first pop.

The identity function serves as an invariant because any stack state should be identical to the new state resulting from making one push followed by one pop operation. The effectiveness of RAAM as a stack representation hinges on how well it has learned to replicate the input in its output. The experiments that we have conducted try to quantify the correlation between approximating the identity function and providing a useful stack.

2.2 The Exponential Decay Model

Because RAAM is simply trained to fit the identity function, there is no explicit representation for the stack state. The network is supposed to discover a compact distributed representation in its hidden layer as a side-effect of learning the target function. We believe that an explicit representation provides two benefits. First, it reduces the search space for adjusting the weights of the network. Knowing the intermediate representation allows us to train the push and pop operations separately so that the functions being learned during training more resemble the tests being performed. Second, knowing the intermediate representation helps us understand the behavior of the network.

2.2.1 Stack Representation

While a stack is usually thought as an array of items, we consider an alternative representation that treats a stack as a set of position lists, each of which is associated with the prototype of an item that can be pushed into/popped from the stack [7]. Under this decentralized object-oriented scheme, the stack state and operators are defined with respect to each position list.

The position list is a binary vector that keeps track of which positions of the stack are currently filled with an instance of its corresponding prototype.

The Exponential Decay Model encodes each position list as a single floating point value, the weighted sum of the positions. As its name suggests, the Exponential Decay Model devotes exponentially more weight to an item at the top of the stack than an item at the bottom. When an item is pushed further down the stack, it loses its weight in the position list exponentially, and thus is easier to be forgotten or confused with its neighbors.

For a concrete example, let us consider a position list that is associated with the symbol 'a.' Let $stack_a$ represent the current stack state (i.e. the current state of the position list for the symbol 'a'). Each state can be mapped into a unique number between 0 and 1 in the following manner:

$$stack_a = \sum_{i=1} \frac{pos_i}{2^{i-1}},$$

where $pos_i \in 0, 1$ indicates whether an instance of the symbol 'a' exists at the ith position of the stack with the top of the stack being pos_1.

Using this representation for the symbol 'a', the push operator takes 2 arguments as inputs: sym_a, a flag to indicate whether 'a' is to be pushed onto the stack, and $stack_a$, the current position list state for 'a.' We express the $Push$ target function as:

$$Push(sym_a, stack_a) = (stack_a + sym_a)/2.$$

Given the current state $stack_a$, the pop operator must provide two values: a binary flag indicating whether the item just popped was the symbol 'a,' and the updated position list state for 'a.' The former is modeled by the function $PopSym$, and the latter by $PopStack$. $PopSym$ can be easily determined by checking whether the weight of the top position contributes to the value of $stack_a$. That is:

$$PopSym(stack_a) = \begin{cases} 0 & : \quad stack_a < 0.5 \\ 1 & : \quad stack_a \geq 0.5 \end{cases}$$

$PopStack$ is slightly more complicated. First, if there is an instance of 'a' on the top of the stack, 0.5 is subtracted from the $stack_a$, representing that the top element has been deleted. Then, it is multiplied by 2 because all instances still inside the stack are now at one position higher than before.

$$PopStack(stack_a) = (stack_a - \frac{PopSym(stack_a)}{2}) * 2 \quad (1)$$

$$= 2stack_a - PopSym(stack_a) \quad (2)$$

This function is the composite of the inverse functions of $Push(sym_a = 1)$ and $Push(sym_a = 0)$.

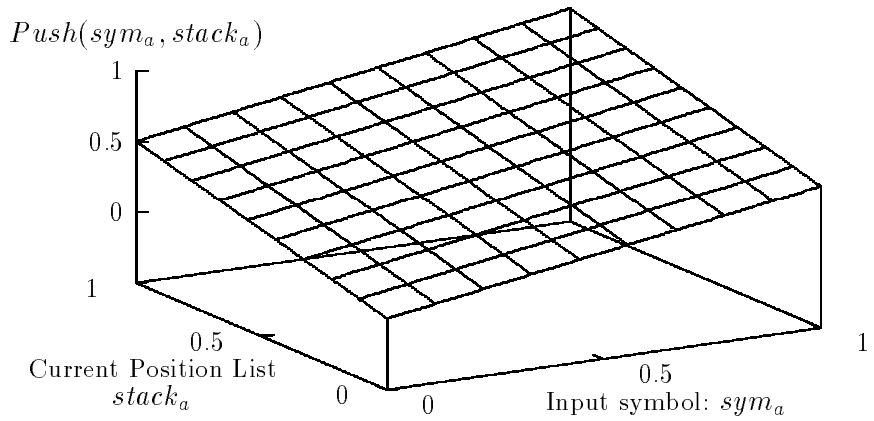

Figure 1: The $Push$ function is a simple plane in 3-D space.

2.2.2 Network Architecture

In designing the network architecture, we must ensure that enough resources are provided to learn the target functions. First, we visualize the $Push$ function in 3-D space. As shown in Figure 1, the target function is a plane, which can be approximated by a simple network of two input units connected to one output unit without any intermediary layers.

The pop phase outputs two functions. First, $PopSym$ is a simple decision function that checks whether the input is greater than 0.5. Again, a minimal configuration of an input/output pair is all we need. The main difficulty lies in learning $PopStack$, the function that updates the position list.

Plotting the return values of $PopStack$ against its input $stack_a$, we get a discontinuous function as shown in Figure 2. Because the output of $PopStack$ relies on the outcome of $PopSym$, it cannot be learned without the introduction of at least one extra hidden layer between the present hidden layer and the output layer. Adding more hidden layers provides the intermediary spaces needed to calculate the output. Later, we shall see how the size of the network affects its performance.

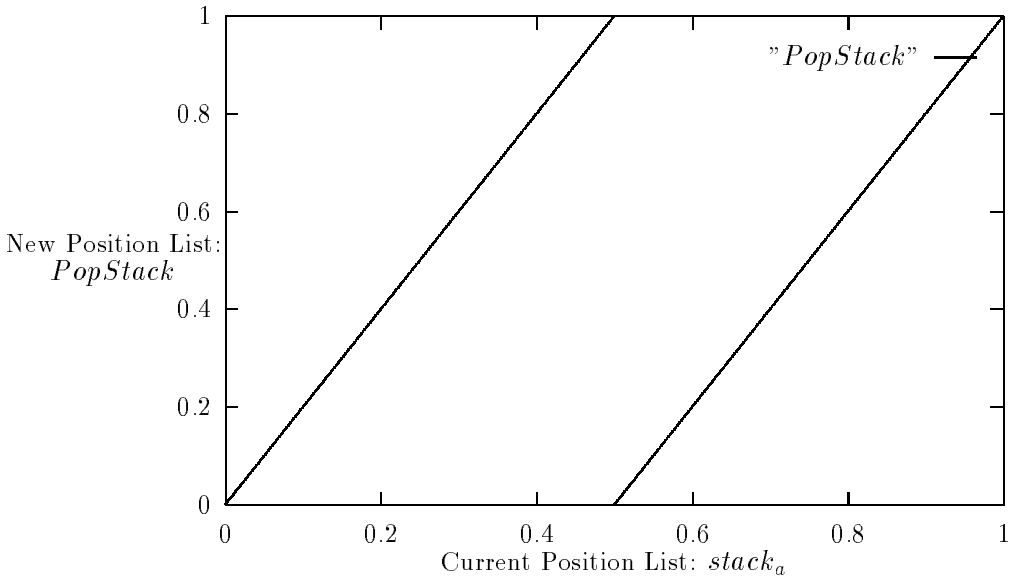

Figure 2: *PopStack* as a function of the current position list $stack_a$.

One plausible configuration for producing *popStack* is shown in Figure 3. Node 1 is the original single unit hidden layer, which represents the current position list. Node 2 outputs *PopSym*; the value is passed along into node 5 and node 8. Because at this point, the result of *PopSym* is still being determined, both potential new values of the position list are calculated in parallel. Node 3 assumes *PopSym* will return true, and node 4 assumes it will return false. In the next layer, nodes 6 and 7 each receives one version of the updated position list as well as *PopSym*. If the outcome of *PopSym* corresponds to the assumption of the position lists passed to them, the values are propagated to the output layer; otherwise, the nodes output 0. Finally, the calculation for *PopStack* is completed in node 9, which adds the values sent by node 6 and 7.

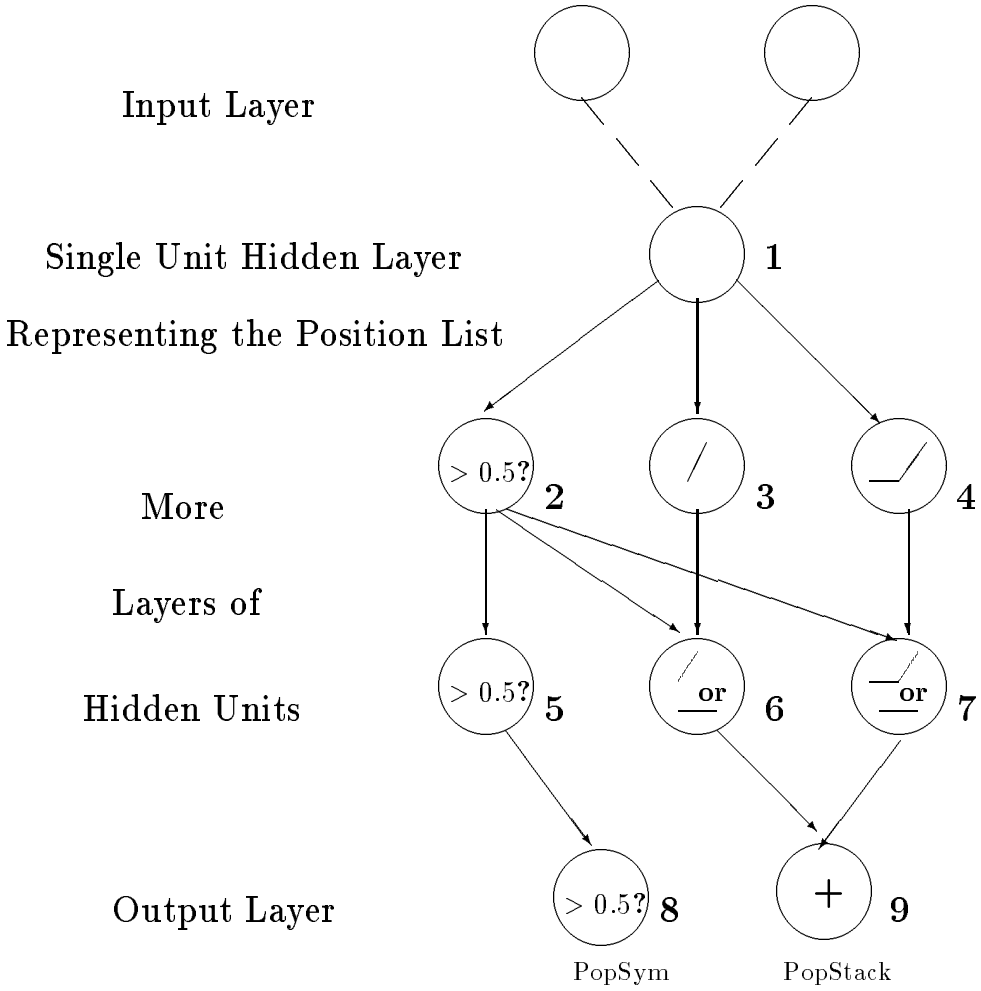

Figure 3: Adding 2 extra layers of hidden units significantly aids the learning of the pop operation.

3 Evaluation Metrics

The models are evaluated by their ability to learn the desired target functions and their performance as stack components of a simple application.

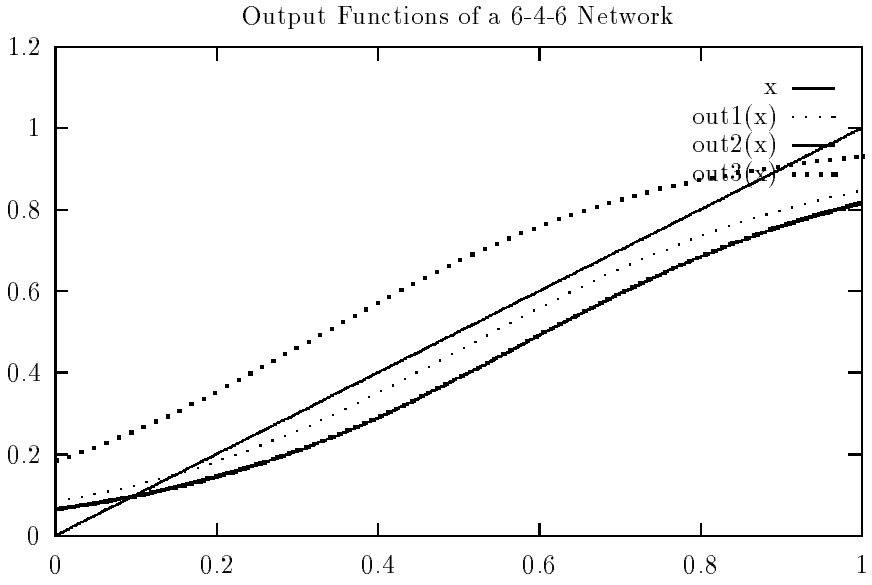

Figure 4: Three output functions of one input node in a 6-4-6 network approximating the identity function. Each output function is generated by a different set of 5 randomly selected real number between 0 and 1 as inputs.

3.1 Learning Target Functions

One fundamental performance metric is to determine how well the models have learned their respective target functions. We empirically show that while both systems have learned to simulate the target functions to some degree, they cannot closely approximate a linear target function because a node in a network is inherently non-linear.

The high dimensional input and output layers of RAAM makes it difficult to visualize the function produced by the network. To see how well the learned function approximates the target function, we randomly set $n-1$ of the input/output units, varying only the n_{th} input over values between 0 and 1. Plotting the n_{th} output unit value against the corresponding input value, we can visualize how RAAM fairs in approximating the Identity Function on a one dimensional level. From Figure 4, we see that the area between the learned function and the target function is not small. Moreover, because the plot shows three sets of averages from multiple sampling, we see that the quality of the learned function in the network is not consistent.

In contrast, the output functions of the Exponential Decay Model can be easily visualized because it maps into a one dimensional space. Figure 8 plots three of the model's learned functions against the target $Push$ function.

3.2 The String Reversal Application

If a model were able to learn the target functions perfectly, it would act as a successful stack. The converse is not necessarily true. How well do the models learn the functions they are trained to learn is, in some ways, tangential to the problem. A more compelling metric is to use the trained networks as stacks in some application. The second experiment we perform is to use each model as the stack component of a string reversal program. The test set consists of several strings of length l, and the strings are of the form $\#[a+b]^l$, where the a's and b's are randomly chosen under a uniform distribution. After all the characters of one string are pushed onto the stack, we can construct the reverse of the string by consecutively popping the stack l times.

Scoring of the models' mistakes is measured by the minimum edit distance [12] between the expected string and the actual one produced by the stack (i.e. the least number of editing operations needed to transform the actual string into the expected string). We have chosen this metric because it reduces the number of over-counting errors. For example, although the strings "abababa" and "bababab" have no matching positions, they are actually very similar because only two edits are needed to transform one into the other. The score for a test set averages the total error count by the number of characters in the test set.

The result of the comparison between RAAM and the Exponential Decay Model is presented in Figure 5. Both models can perfectly reverse strings of length four or shorter, but the Exponential Decay Model's performance degenerates faster than that of RAAM.

4 Relevant Parameters

The performance of a neural network can be affected by several parameters. In section 4.1 we experiment with different training strategies; and in section 4.2 we vary the number of hidden units in the network. Different training strategies makes little differences for both models, but RAAM is more sensitive to the variation in the size of the network.

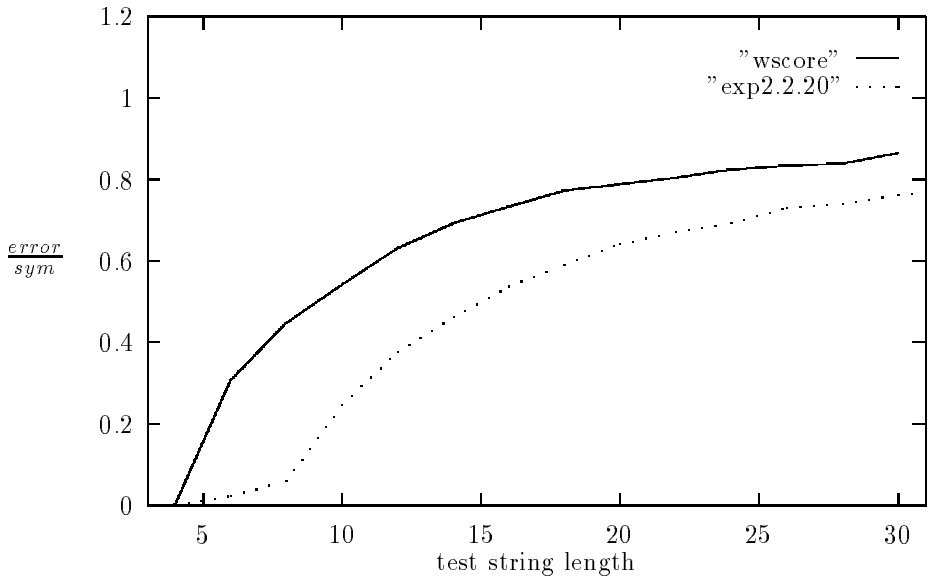

Figure 5: A comparison of RAAM and the Exponential Decay Model performing the string reversal application

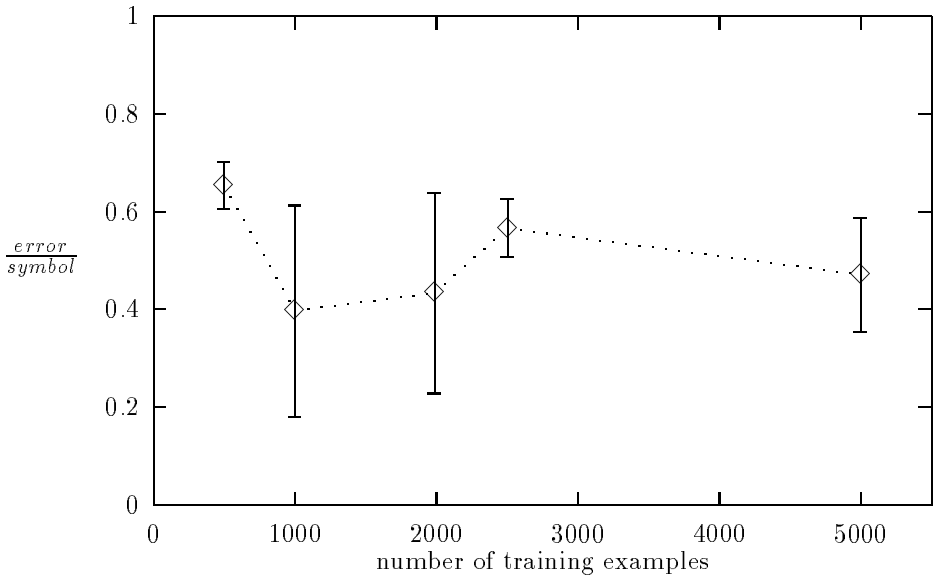

Figure 6: Error rate as a function of training set size s

4.1 Training Strategies

Because RAAM has too many input units to be trained through uniform sampling, Pollack trains his network by using the initial hidden layer values as part of the training environment. That is, he starts with some initial input pattern, which generates some hidden layer pattern, which is then used as the next input pattern. (Pollack observed that the convergence of the network for this form of "moving target" learning is dependent on the learning parameters of the network: the learning rate and the momentum. For these learning parameters, we used the values he suggested.) In other words, the network is continuously pushing symbols of one long string of length s onto the stack. To determine the effect of the size of the training data, we vary s. Figure 6 shows the error rate for four different sizes of s: 496, 1000, 1984, 2500, 5000. We find that the performance tends to improve only slightly when given more training data.

Under Pollack's training scheme, the system may be optimized for one sequence at the cost of other strings. For the stack application it seems that better performances might be achieved if the distribution of stack states is taken into account. More specifically, about half the strings in the test set

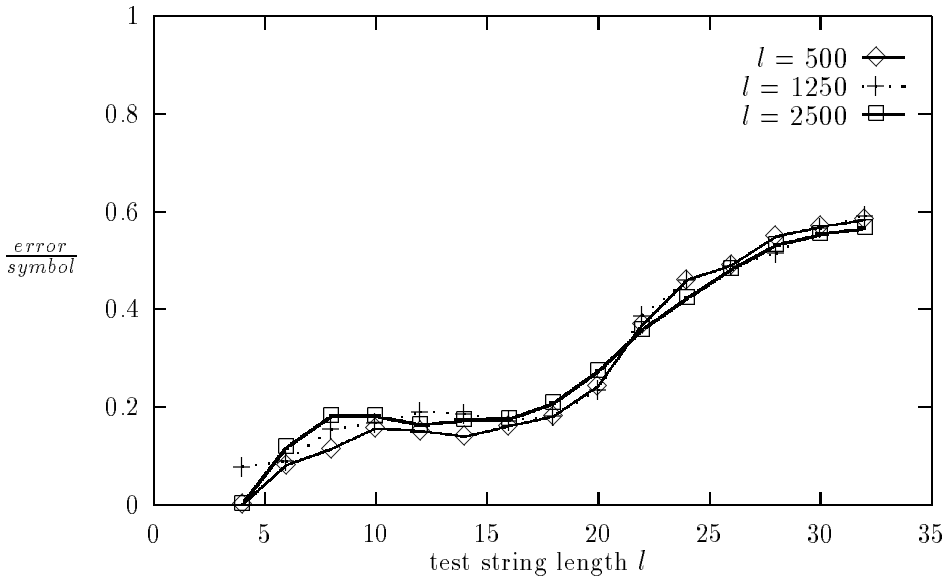

Figure 7: performance comparisons between ratios (500:5), (1250:2) and (2500:1)

will begin with "#a" and half with "#b." About 25% of the strings starts with "#aa," and so forth. To take into the distribution of the stack states, we varied Pollack's method by training the network on k strings of length l, where $k * l = m$. To show the effects on the network's performances when it was trained with different $l : k$ ratio, we choose a training set of size $m = 2500$, and trained a network with 16 hidden units under three ratios – (500:5), (1250:2), and (2500:1). From Figure 7 we see that varying the lengths of the training sequences does not have a significant effect on the performance of RAAM.

To uniformly sample the target functions in the Exponential Decay Model, we can afford to pick i equally spaced numbers between 0 and 1, and train the network on a training set of size $s = i^M$ examples. We have tried four different sizes of s: 100, 200, 500, and 1000. In Figure 8, we compare the functions learned by the network for the four values of s. As one might expect, the bigger s is, the closer the network learns $Push$, but the differences are small.

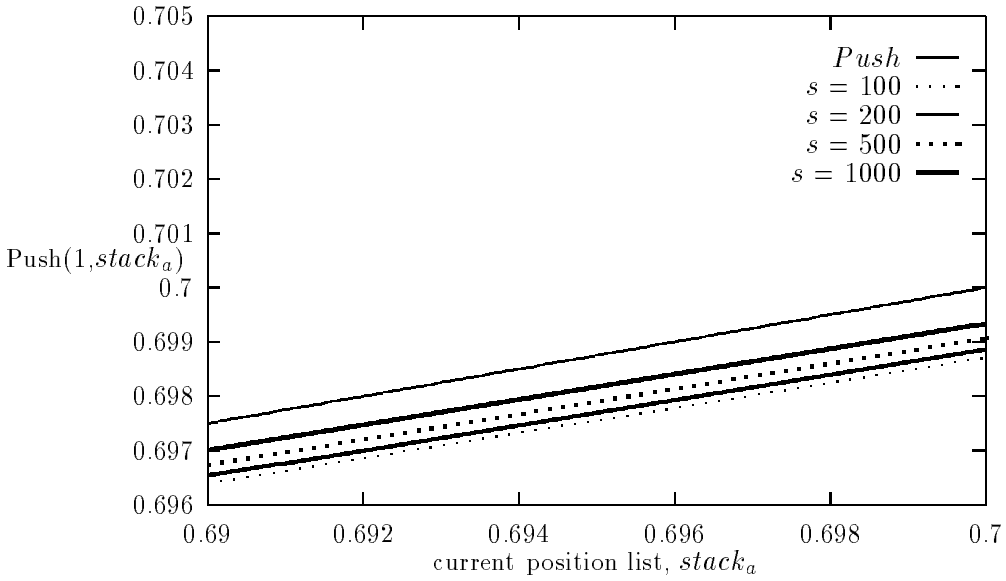

Figure 8: The function $Push$ when $sym_a = 1$

4.2 Number of Hidden Units

We can also vary the size of the network for both models. We have discovered that the performance of RAAM fluctuates more with different network sizes than the Exponential Decay Model. We have trained 16 RAAM networks with different number of hidden units (4 to 32 inclusive, at even intervals). All the networks use the same training set, each consisting of 1000 characters. As evident from Figure 9, the average error made per symbol tends to decrease as the number of hidden units increases, although the improvements are not as significant as the increase in training time.

In the Exponential Decay Model, $PopStack$ is the most difficult function to learn . Based on the reasoning behind figure 3, the network uses two extra hidden layers. We have varied the number of units per hidden layer between 3 and 27 in increments of two. The network is given 500 pairs of input output pairs of $PopStack$. We approximate the mean square error by summing over the square error of the testing points. Figure 10 plots the total sum of squared errors of each network. Although the error rate decreases when more hidden units are added, when the networks are tested on the string reversal applications, there is little difference in performance.

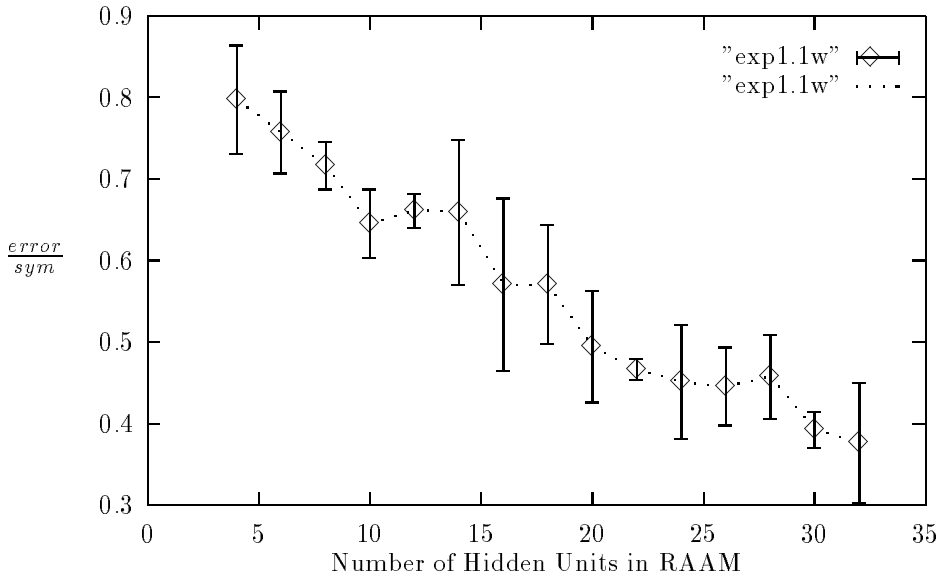

Figure 9: As the number of hidden units increases, the error decreases

5 Discussion

From all the experiments we have described so far, it seems that a model's ability to learn the target functions is not a complete indicator as to whether it will perform well as a stack, nor is the intrinsic differences between the two models' philosophy of representation. Both the implicitly learned internal stack state representation of RAAM and the explicitly trained stack state representation of the Exponential Decay Model cannot overcome the rapid rate of error propagation inherent in recursive networks. Even the smallest deviation in the learned function from the target function can cause the quality of the stack to degrade drastically within a couple of iterations.

What does seem to play a larger role in determining the speed of deterioration in the models is the number of dimensionality (and therefore the size of the network). Increasing the size of RAAM helps its performance because the dimensionality of the network is also increased, which allows it to store more information about the training data. For the Exponential Decay Model, on the other hand, the input and output are so firmly rooted in a single dimension representation that by simply increasing the

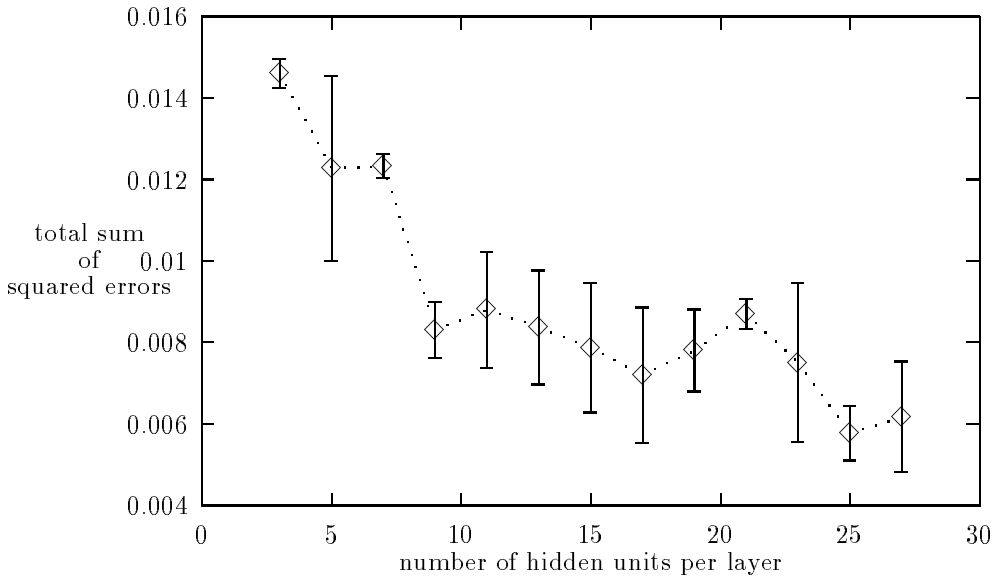

Figure 10: The number of hidden units affects the network's ability to learn *PopStack*

number of internal nodes did not help its performance very much. There is, however, a limit to which increasing the dimensionality of RAAM can help. Because the size of the training data required grows exponentially as the size of the network increases, we cannot realistically increase the ordinality of dimensions beyond 20.

6 Conclusion

In this paper, we have compared two representations of the stack data structure: RAAM and the Exponential Decay Model. Through empirical studies, we have shown that the trained networks of both approaches can, to some degree, behave like a stack, but the stack size is very shallow. Although the models use different representations, they are limited by the recurrent and non-linear nature of their network architectures.

References

[1] Berg, G. "A connectionist parser with recursive sentence structure and lexical disambiguation," *Proceedings of the 10th National Conference on Artificial Intelligence,* 32-37. Cambridge, MIT Press, 1992.

[2] Charniak, E. and Santos, E. "A connectionist context-free parser which is not context-free but then it is not really connectionist either," *Proceedings of the Ninth Annual Conference of Cognitive Science Society*, pp 70-77, 1987.

[3] Cybenko, G. "Continuous-valued neural networks with two hidden layers are sufficient." (unpublished), 1988.

[4] Elman, J. L. "Finding Structure in Time," *Cognitive Science*, vol. 14, pp. 179-211, 1989.

[5] Fanty, M. "Context-free parsing in connectionist networks," *Computer Science Department of the University of Rochester Technical Report TR174*, 1985.

[6] Jain, A. N. "Parsing complex sentences with structured connectionist networks," *Neural Computation,* Vol. 3, pp. 110-120, 1991.

[7] Ketchpel, S. "A Connectionist Approach to Parsing," (unpublished), 1992.

[8] McClelland, J. L. and Rumelhart, D. E. *Explorations in Parallel Distributed Processing,* Cambridge, MIT Press, 1989.

[9] Miikkulainen, R. and Bijward, D. "Parsing embedded clauses with distributed neural networks," *Proceedings of the 13th National Conference on Artificial Intelligence,* pp. 858-864, 1994.

[10] Pulman, S. G. "Grammars, parsers, and memory limitations," *Language and Cognitive Processes*, Vol. 1, No. 3, pp. 197-225, 1986.

[11] Pollack, J. "Recursive distributed representation," *Artificial Intelligence*, Vol 46, pp. 77-105, 1990.

[12] Wagner, R. A. "The string-to-string correction problem," *Journal of the Association for Computing Machinery*, Vol. 21, No. 1, pp. 168-173, 1974.