Connection Science
Vol. 00, No. 00, Month 2011, 1–25

RESEARCH ARTICLE

Processing of Nested and Cross-Serial Dependencies: an Automaton Perspective on SRN Behaviour

Christo Kirov[a*] and Robert Frank[b]

[a]*Cognitive Science, Johns Hopkins University, MD, USA*; [b]*Linguistics, Yale University, CT, USA*

(*v3.0 submitted November 2011*)

Language processing involves the identification and establishment of both nested (stack-like) and cross-serial (queue-like) dependencies. This paper analyzes the behavior of Simple Recurrent Networks (SRNs) trained to handle these types of dependency individually and simultaneously. We provide new converging evidence that SRNs store sequences in a fractal data structure similar to a binary expansion. We provide evidence that the process of recalling a stored string by an SRN depletes the stored data structure, much like the operations of a symbolic stack or queue. Trained networks do not seem to operate like random access arrays, where a pointer into a data structure can retrieve data without altering the contents of the data structure. In addition, we demonstrate that networks trained to model both types of dependencies do not implement a more complex, but unified, representation, but rather implement two independent data structures, similar to a stack and queue.

Keywords: SRNs; Fractal Representations; Nested and Cross-Serial Dependencies; Multi-Task Networks; Network Implementation of Symbolic Structure

1. Introduction

Language processing involves the identification and establishment of non-local dependencies among the elements in the sequence of linguistic forms. In *center-embedding*, these dependencies have a nested character, with the first element dependent on the last one, the second element on the penultimate, etc. Center embedding is found in the syntax of a wide variety of human languages, the following example from English being one case:

(1) The book(1) that someone(2) I(3) know(3) wrote(2) got reviewed(1) in the *NY Times*.

In this sentence, the first noun phrase *the book* is semantically dependent on the last verb *got reviewed*, the second noun phrase *someone* is dependent on the second-to-last verb *wrote*, and the third noun phrase is dependent on the third-to-last verb *know*. In symbolic computation, the processing of center-embedded constructions is easily characterized through the use of a *stack*. A stack is a first-in, last-out (FILO) data structure where items are retrieved (*popped*) from a stack in reverse order from when they were put in (*pushed*). The example above would be processed by first pushing the three noun phrases onto a stack resulting in a data structure like [$\perp$, *the book*, *someone*, *I*], where $\perp$ represents the bottom of the stack. As each verb

*Corresponding author. Email: kirov@cogsci.jhu.edu

ISSN: 0954-0091 print/ISSN 1360-0494 online

DOI: 10.1080/09540090xxxxxxxxxxxx
http://www.informaworld.com

is encountered, the element at top of the stack is popped off and matched with the verb, until the stack is empty. Thus, upon encountering *know*, *I* is removed from the top of the stack to form the relevant semantic dependency, leaving the stack [⊥, *the book*, *someone*]. This process would be repeated with the second verb *wrote*, popping *someone* from the stack, and finally *the book*, the first element that was inserted, would be popped to form the association with the verb *got reviewed*, leaving an empty stack. In addition to accounting for center-embedding in human language, a stack also provides a natural account of performance on a reverse serial recall task, where human memory performance is tested by having subjects repeat a list of symbols they have just seen in reverse order (Li and Lewandowski 1995).

Cross-serial dependencies are a second, though somewhat less common, strategy for forming non-local relationships in human language. In such cases, the dependent elements appear in a crossing rather than nested order, so that the first element in one part of a sequence is dependent on the first element in another part, the second element on the second, and so on. An example of this is found in Dutch (Bresnan et al. 1982, Huybregts 1976):

(2)

. . . omdat	*ik(1)*	*Cecilia(2)*	*Henk(3)*	*de*	*nijlpaarden*	*zag(1)*	*helpen(2)*	*voeren(3).*
. . . because	I	Cecilia	Henk	the	hippo	saw	help	feed

'because I saw Cecilia help Henk feed the hippo.'

In this case, the first noun phrase *ik* is semantically dependent on the first verb *zag*, the second noun phrase *Cecilia* on the second verb *helpen*, and so on. Although not as prominent as in Dutch, English also shows constructions that can be taken to involve cross-serial dependencies: in VP-ellipsis, a verb phrase that is omitted is understood to be an in-order copy of a verb phrase that occurred earlier (Stabler 2004). Such a copying relationship can be represented as a cross-serial dependency.

(3) John didn't give(1) the presents(2) to Mary(3) yesterday, but I think Bill did ~~give(1) the presents(2) to Mary(3)~~ today.

The natural symbolic data structure for characterizing cross-serial dependencies is a *queue*. A queue is a first-in, first-out (FIFO) data structure, where items are retrieved in the same order they are put in. So, the Dutch example above would be processed by first filling a queue with each subject noun phrase of the sequence [*ik*,*Cecilia*, *Henk*], then retrieving each stored noun phrase from the queue in order to match with each verb of the sequence. Queues are also a natural fit for describing the forward serial recall task (the reverse of the reverse recall task described above), where subjects are asked to repeat a list of symbols they have just seen while preserving the order of the list (Page and Norris 1998, Henson 1998).

Since human language exhibits both types of dependencies, linguists have tried to identify formalisms that are sufficiently rich to represent both center-embedding constructions and cross-serial dependencies. In the hierarchy of grammars identified by Chomsky (1959), the least powerful system capable of representing both types is Type-1 or Context Sensitive Grammars. Such grammars are extremely expressive and do little to limit the class of possible human languages. Given the existence of universal constraints on languages, and the fact that certain types of potential grammatical patterns do not actually occur, it is widely accepted that context-sensitive grammars are insufficiently restrictive to be the correct formalism for describing human language grammars. Mildly Context Sensitive formalisms, including Tree Adjoining Grammars are considerably more restrictive than Context Sensitive Grammars, yet powerful enough to characterize the constructions found in human languages, including both center embedding and cross-serial dependencies (Joshi 1985). The computational realization of these grammatical formalisms makes use of stacks and queues. For example, Embedded Push Down Automata, the automaton equivalent of Tree Adjoining Grammars, store data as a stack of stacks (Joshi 1990, Joshi and Schabes 1997).

It has been widely observed that human processing of sequences involving center-embedding and cross-serial dependencies is sharply restricted. For example, with more than 3 levels of dependencies, sentences like the center-embedding case given earlier become virtually incomprehensible.

(4) The book that someone a teacher I know told me about wrote got reviewed in the *NY Times*.

Miller and Chomsky (1963) suggest that such anomaly is best explained in terms of a bound on the depth of the stack used for language processing, i.e., as a kind of memory limitation. Such a restriction on stack depth has struck many as a arbitrary and incapable of dealing with the variation in processing difficulty that results from changes in the fine-grained properties of the dependent elements. These problems, and the apparent lack of connection between symbolic formalisms and human neural processing, has caused some researchers to explore connectionist models of stack-like and queue-like processes and their role in language processing (Christiansen and Chater 1999). One particular promising connectionist model of language processing is Simple Recurrent Networks, or SRNs (Elman 1990). SRNs are a type of recurrent neural network where processing at a particular timestep depends not only on the current input but also on the internal state of the network at a previous timestep, stored in a set of so-called context units. In other words, SRNs maintain a history of old computations which affect new computations. SRNs were devised as a means for connectionist networks to process temporally separated sequences of elements, such as the words of a sentence. In the most common scenario, an element from the sequence is presented as input to the network, and the target output is the next element in the sequence. As success in predicting the next input depends on knowledge of the structure of the sequence, this task provides a way of assessing a network's success at learning to process sequences of different types.

SRNs have been studied for their ability to process sequences that exhibit long-distance dependences of a variety of sorts. Servan-Schreiber et al. (1991) trained SRNs to perform prediction in the case of a language in which the first and last symbols were identical, but were separated by an unboundedly long string drawn from a finite-state language. Wiles and Elman (1995) demonstrated that SRNs could represent a simple counting language $a^n b^n$, suggesting that they have the ability to use either a stack or a queue: by pushing the as from the first part of the string into either a stack or a queue, each a can be checked against a b in the second portion to confirm that there is an appropriate correspondence.[1] Later work by Tabor (2000), Rodriguez (2001), and Grüning (2006) showed that SRNs could handle languages with center-embedded dependencies. For instance, SRNs can be trained to recognize a palindrome language, i.e., one including only strings of the form $ABCD\#dcba$, indicating their ability to simulate the effects of processing with a stack. Similarly, SRNs have been shown to be able to recognize languages with strings of the form $ABCD\#abcd$. Such languages, with their cross-serial dependencies, require queue-like operations for their processing, where the second

[1]Having a single counter is like a limited stack or queue which can only store one type of symbol. Unlike with a stack or queue with a larger alphabet, it does not matter which end of the data structure we read from; only the number of stored symbols matters.

half of the string is recognized as the removal of items from a queue in first-in first-out order (Grüning 2006). These simple formal languages capture the essence of the center-embedding and cross-serial dependencies seen in human languages, and the memory performance required for human serial recall tasks. Thus, researchers have also explored SRNs as models for human performance of sentence processing (Christiansen and Chater 1999, Badecker and Frank 2008, Frank et al. 2008) and serial recall (Botvinick and Plaut 2006, Goldberg and Rapp 2008). Interestingly, it turns out that beyond a certain limit, increasing network complexity in the form of extra hidden units does not improve performance on the processing of non-local dependencies. This is in contrast with the symbolic approach, where performance can be improved simply by increasing the amount of memory available. Consequently, Christiansen and Chater (1999) have claimed that the limitations of trained networks on the tasks above are intrinsic to the SRN architecture. It should be noted that this may be a limitation of the learning algorithm used to automatically train the networks, as Siegelmann and Sontag (1995) showed that finite recurrent neural nets, of which SRNs are a subset, can be made to simulate any Turing machine given some appropriate set of hand-coded weights.

Previous work has focused on the performance of networks trained to act *either* as a stack or as a queue. Within a single language, however, we can find evidence for both kinds of dependencies. For instance, because of the possibility of verbal extraposition, German permits both center-embedded and cross-serial orderings in examples like the following:

(5)	. . . *weil*	*ich(1)*	*das*	*Fahrrad(2)*	*zu*	*reparieren(2)*	*verspreche(1)*	
	because	I	the	bicycle	to	fix	promise	

‘because I promise to fix the bicycle.’

(6)	. . . *weil*	*ich(1)*	*das*	*Fahrrad(2)*	*verspreche(1)*	*zu*	*reparieren(2)*	
	because	I	the	bicycle	promise	to	repair	

As a result, a model of sentence processing must be able to process dependencies of both sorts within a single language. In this paper, we investigate the viability of SRNs to do this. We explore the performance of networks trained to act as stacks *and* queues, switching between the two different modes of operation depending on the input. We pay particular attention to diagnosing the type(s) of data structure(s) a network trained to process both dependency types might use, and to determining the extent to which these dependency types are processed independently of one another.

The paper is structured as follows. After briefly describing the SRN architecture, we present an idealized picture of how SRNs might operate as stacks and/or queues, both independently and together. Storage and recall are reduced to manipulation of a binary expansion, or, equivalently, a fractal tree. After describing the various ways networks might actually operate using this formalism, an empirical analysis of trained network behavior is used to weed out which possibilities are more likely. Unlike in the symbolic formalisms mentioned above, the representations and algorithms used to perform a given task are not explicitly known for an SRN. It has been noted that the performance of trained networks is sometimes difficult to describe analytically, and that networks can therefore be treated as objects of experimental study similar to brains (McCloskey 1991). The empirical portion of this paper applies this approach, using network behavior to distinguish different theories of how the networks perform their task. In addition, because of our explicit hypotheses concerning the connectionist implementation of symbolic structures, we are able to move beyond simply measuring success in the task. Specifically, we assess the degree to which the patterns of hidden unit activation in the trained

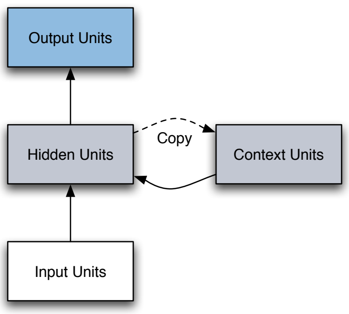

Figure 1. Connectivity structure of first order SRN.

network reflect the kinds of patterns our neurally-implemented symbolic models would lead us to expect. In this respect, we are following the line of work begun by Servan-Schreiber et al. (1991), who explored the degree of fit between hidden unit activations and finite state automata.

2. SRN Architecture Overview

The first order SRN (Elman 1990) has an input layer of units, a hidden layer, a context layer that stores the values of the hidden layer from the previous time step, and an output layer. It can be described by the following equations.

$$H_t = F(W_{HH} \cdot H_{t-1} + W_{HI} \cdot I_t + b) \tag{7}$$

$$O_t = W_{OH} \cdot H_t \tag{8}$$

H_t represents the vector of hidden units at time t, I_t represents the vector of input units at time t, and O_t represents the outputs of the network. At each time step, the hidden units are updated by multiplying the input units and context units by a trainable weight matrix, and passing each element of the vector representing the sum of these two multiplications and a bias term b through a standard sigmoid function, $y = \frac{1}{1+e^{-x}}$. F in the equations above represents the point-wise application of a sigmoid function to each value in the hidden unit vector. Since the computation at each time step uses the previous state of the network in the form of the context units, it is said to have a context, or memory of the network's previous computations. Activation on the output units is computed by multiplying the hidden unit activations by a third trainable weight matrix.

3. Idealized Networks

Throughout this paper, we attempt to shed light on a few basic questions regarding the high level strategies discovered by SRNs that are trained to perform forward and backward serial recall. First, how do networks that act as either a stack or a queue manage their internal data structures? For example, in the traditional symbolic implementation of a stack or a queue, reading from the data structure removes a symbol from the stored string of symbols, making it unavailable for subsequent reads. Do SRNs implement stacks and queues in the same way? Does reading a symbol from a stored string deplete the network's internal data structures? Second, if a network is trained to act as both a stack and a queue, does it perform these two tasks independently? Do both tasks share the same data structure?

We can lay out the space of possible strategies networks may use for representing stacks and queues by focusing on the possible variations of a formalism initially proposed by Grüning (2006) to describe the operation of SRNs. In this formalism, an SRN stores sequences in a format equivalent to a binary expansion of a real number in base 2. In this paper, we use a binary expansion since we only consider languages over the alphabet $\{0, 1\}$ for simplicity. However, the construction readily generalizes to an n-ary expansion in the case of larger alphabets of symbols. For example, the sequence 0101 can be represented as the binary number 0.0101, which is equivalent to the decimal value $0+1/4+0+1/16 = 0.3125$.[1] Note that, in an actual SRN, the binary expansion and other related variables described here would be represented in the network's hidden units, but they need not be directly represented in the value of a single hidden unit. They may instead have a distributed representation that spans many hidden unit values. This is loosely analogous to how a single floating point number is stored in a computer using several bits. The discussion here abstracts away from the exact structure of hidden unit representations.

Grüning describes two sets of equations based on this formulation that, if implemented by an SRN, would allow it to act as a stack and perform backward recall or as a queue and perform forward recall. We will examine the case of backward recall first, since it is simpler.

We maintain a stack variable x. x is a real-valued variable that stores the binary expansion. The key idea is that every time we see a 0 or 1 in the input during encoding, we place it in the most significant slot in the expansion. In order to do this we put the symbol we are encoding just to the left of the decimal place in the expansion via addition, and then divide our expansion by 2, shifting it once to the right. Conveniently, we can retrieve any previously seen symbols by shifting the expansion back to the left and reading off the symbol in the most significant slot before discarding it. The equations implementing this scheme are shown in (9).

(9) Equations for performing backward recall

Input	Internal Representations	Output
init	$x = 0$	
0	$x \rightarrow (x + 0)/2$	
1	$x \rightarrow (x + 1)/2$	
recall cue	$x \rightarrow 2x \bmod(1)$	$2x \geq 1$

[1]Botvinick and Plaut have proposed that trained SRNs use a "conjunctive code". Each symbol and position pair (e.g. a 0 at position 3 in the string) is represented by a unique vector of hidden unit activations. An entire string is formed by adding, or superimposing, a set of these vectors together. For example, the string "010" would be represented by hidden unit activations as

$$010 = 01 + 12 + 03$$

where 01 is the conjunctive code for '0' in position 1, 12 is the conjunctive code for '1' in position 2, and 03 is the conjunctive code for '0' in position 3. The n-ary expansion described above is an instance of such a code, as the expansion can be broken down into a sum of symbol*position codes. For example,

$$0.010 = 0 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3}$$

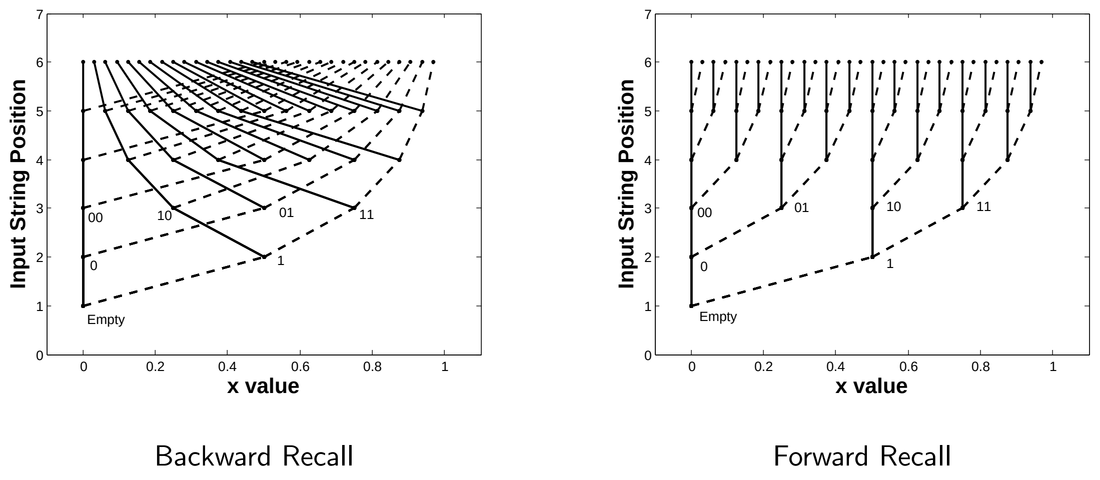

Figure 2. **Right**: Fractal encoding pattern generated by storage using the backward recall equations. **Left**: Fractal encoding pattern generated by storage using the forward recall equations. In both figures, solid branches indicate 0 as the next input symbol, while dashed branches indicate 1 as the next input symbol. The nodes in the fractal tree correspond to encoded strings, where the x axis value is the encoding, and the y axis value is the length.

These equations show how the network manipulates its internal state in response to different inputs. The output of the network during recall is given as a condition (or hyperplane, in network terms) over its internal representations. Thus if $2x \geq 1$ is true, the network outputs 1. If not, the output is 0. Encoding and recalling strings using these equations can be visualized as following the branches of a fractal structure (Tabor 2000, Bodén and Blair 2003). In the backward case, the fractal generated is shown in Figure 2 (Right).

The forward recall case is a bit more complicated. For an input string $x_0x_1\dots x_n$, we are looking to generate an expansion of the form $0.x_0x_1\dots x_n$. This means that we cannot keep inserting new symbols into the most significant slot as we did in the backward recall case. Each symbol must be inserted into the *next* most significant slot after the previous symbol. We achieve this by having an extra variable y act as a way of keeping track of what the appropriate slot to insert the next symbol is (i.e. what 2^{-n} to multiply the symbol by before adding it into the expansion. This scheme allows us to use the exact same retrieval procedure we used in the backward recall case. The equations are shown in (10).

(10) Equations for performing forward recall

Input	Internal Representations	Output
init	$x = 0,\ y = 1$	
0	$x \rightarrow x + 0 * (y/2),\ y \rightarrow y/2$	
1	$x \rightarrow x + 1 * (y/2),\ y \rightarrow y/2$	
recall cue	$x \rightarrow 2x \,\mathrm{mod}(1)$	$2x \geq 1$

Once again, we can generate a fractal tree using these equations, as shown in Figure 2 (Left).

In Grüning's formulation, stack and queue networks use different encoding strategies (stacks insert new symbols into the most significant slot of a binary expansion, queues use the least significant slot) but identical recall strategies. In addition, the recall step alters the x variable. This depletes the data structure where the input string is stored, as after the mod(1) operation the last recalled symbol can no longer be recovered.

Depletion is not necessary if we allow the y dimension to serve as a pointer into the stored binary expansion. This would allow stack and queue networks to fill using either the most significant or least significant digit of the expansion during encoding. Again, a variable y would serve as a counter for the depth of the string. During recall, this counter would serve as a multiplier for the binary expansion, indicating how far it needs to be shifted such that the digit to the right of the decimal point was the digit to be output. The x variable is not changed as a result of this recall operation, and so the data structure storing the input string is not depleted. After the output value is read, only the pointer itself will be either divided or multiplied by 2 to point to the next symbol for the next time step of recall. For example, the equations in (11) act like a queue despite putting each successive symbol of the encoded sequence into the most significant slot of the expansion like Grüning's stack equations.

(11) Forward recall equations using a pointer during recall

Input	Internal Representations	Output
init	$x = 0,\ y = 2$	
0	$x \rightarrow (x + 0)/2,\ y \rightarrow 2y$	
1	$x \rightarrow (x + 1)/2,\ y \rightarrow 2y$	
recall cue	$y \rightarrow y/2$	$yx \bmod(2) \geq 1$

Sticking with the idea of using binary expansions as data structures, how could a network faced with learning to act as both a stack and a queue work? The simplest strategy would be to encapsulate the functionality of two independent stack and queue networks. Each subnetwork would have its own expansion (x) and counter (y) variable, which would be updated at each encoding step. During recall, receiving a forward or backward cue would tell the main network to call the appropriate subnetwork as a function. This fully independent setup is shown in (12).

(12) Equations for forward and backward recall using independent representations

Input	Internal Representations	Output
init	$x_1 = 0,\ x_2 = 0,\ y = 1$	
0	$x_1 \rightarrow (x_1 + 0)/2$	
	$x_2 \rightarrow x_2 + 0 * (y/2),\ y \rightarrow y/2$	
1	$x_1 \rightarrow (x_1 + 1)/2$	
	$x_2 \rightarrow x_2 + 1 * (y/2),\ y \rightarrow y/2$	
backward recall cue	$x_1 \rightarrow 2x_1 \bmod(1)$	$2x_1 \geq 1$
forward recall cue	$x_2 \rightarrow 2x_2 \bmod(1)$	$2x_2 \geq 1$

Again, there are alternative plausible strategies for combining forward and backward recall. For example, a combination network could store a sequence in a single x variable, but maintain two different y variables to act as pointers for accessing this variable in the manner described above. The equations in (13) show how we can get both stack and queue behavior depending on recall cue using this strategy. Note that the backward pointer y_1, in this case, is not touched during encoding. Furthermore, the stored data structure x, is not altered during recall.

(13) Equations for forward and backward recall using independent pointers

Input	Internal Representations	Output
init	$x = 0,\ y_2 = 1,\ y_1 = 2$	
0	$x \to (x + 0)/2,\ y_2 \to 2y_2$	
1	$x \to (x + 1)/2,\ y_2 \to 2y_2$	
forward recall cue	$y_2x \bmod(2),\ y_2 \to y_2/2$	$y_2x \bmod(2) \geq 1$
backward recall cue	$y_1x \bmod(2),\ y_1 \to 2y_1$	$y_1x \bmod(2) \geq 1$

Both of these possibilities suggest that networks trained to perform as both stacks and queues perform the two tasks independently, in the sense that recall performed in one direction does not in any way alter the results of recall performed in the other direction. In the first case, the independence is overt, as the network contains separate stack-like and queue-like subnetworks that don't interact. In the second case, both tasks share the same data structure, but they maintain independent pointers into that data structure, and the data structure itself remains untouched throughout recall. Task independence, however, is not a requirement for multi-task networks. It is possible to design such a network so that each subsequent recall effectively depletes an underlying data structure, much as popping from a stack or reading from a queue would do in their canonical symbolic forms. In such a situation, a series of reads from the end of a stored sequence using stack recall could leave the stored data structure empty, so that reads from the front of the sequence using queue recall would have nothing to return. A set of equations that display such behavior are presented in (14). In the approach shown, the symbol to be recalled is set to zero after being output. When the symbol is later read from the reverse direction, zero will always be the output, leading to about 50% error since the symbol was just as likely to be a one originally. Non-independence of the form described here, involving data structure depletion, entails data structure alteration during recall.

(14) Non-independent equations for forward and backward recall

Input	Internal Representations	Output
init	$x = 0,\ y_2 = 1,\ y_1 = 2$	
0	$x \to (x + 0)/2,\ y_2 \to 2y_2$	
1	$x \to (x + 1)/2,\ y_2 \to 2y_2$	
forward recall cue	$x \to x - (y_2x \bmod(2) - y_2x \bmod(1))/y_2,\ y_2 \to y_2/2$	$y_2x \bmod(2) \geq 1$
backward recall cue	$x \to x - (y_1x \bmod(2) - y_1x \bmod(1))/y_1,\ y_1 \to 2y_1$	$y_1x \bmod(2) \geq 1$

We are now in a position to examine the question of task independence empirically through the performance of trained networks.

4. Empirical Examination of Network Performance

Networks were trained to perform several variants of the forward (queue-like) and backward (stack-like) serial recall tasks. Processing of a particular input string was split into two phases: an encoding phase and a recall phase. During the encoding phase, a network was fed one symbol of the input string per time step. As we only considered binary strings there were only two possible inputs during encoding, which we will label 0 and 1. During the encoding phase, the network only had to memorize the input string. Network output during any step of encoding was irrelevant to the overall task. We encode this unimportant output using a special null symbol N.

After the network had seen the entire input string, the recall phase of processing began. During each time step of this phase, the network was given one of two recall cue symbols. The forward recall cue, which we label F, told the network to output the next available symbol from the input string, reading from the front. For example, if the input string was 1010, an F cue during the first time step of recall should cause a network to output the first 1 in the input string. The backward cue, which we label B, told the network to output the next available input symbol, reading from the back. Again, using our example input 1010, a B cue during the first time step of recall should cause the network to output the last 0.

All the networks trained had the same structure, with layout identical to that shown in Figure 1. Each had 20 hidden units.[1] Inputs and outputs were represented using a 1-in-k encoding scheme. Each unique input and output symbol was mapped to a particular unit. When a particular symbol was passed to the network as input, its corresponding unit's activation was set to 1 and all other units were set to 0. There were four input units, corresponding to the two symbols to be stored (0 and 1) and the two types of recall cues, forward and backward (F and B). There were three output units, one for the null output symbol (N) and two for symbols to be recalled (0 and 1). There was a bias unit with a fixed value of -1.

During each training epoch, the network was fed 500 randomly generated training examples. All unit activations were reset before each new example was processed. Sequences to be stored by the network ranged in length from 1 to 9 symbols. Following Rodriguez (2001), training examples were drawn from a random distribution that favored shorter input sequences; for sampling, the lengths 1 though 9 were weighed by an exponential distribution with mean 4.5. Weights were updated after each training example using simple recurrent backprop through time (SRBPTT).[2] Output error was calculated using the standard Euclidean loss function $\frac{1}{2}(O-T)^2$, where O is the network's output and T is the target output value. The learning rate was set at a constant $\eta = .01$, and no momentum was used.

After each training epoch, the network was tested on 100 randomly generated test examples, drawn from the same distribution as the training data. Accuracy was measured as the percentage of test sequences correctly recalled. Correct recall meant that each encoded symbol was reproduced in the output at the correct timestep. At a particular timestep, the recalled symbol was taken to be the one whose corresponding output unit was most highly activated. Networks were trained until they had achieved 80% accuracy during the test phase or 300 epochs had passed. In practice, all trained networks reached the desired accuracy level before the 300 epoch cutoff.[3]

[1]As stated above, there seem to be limits on the amount network performance can be improved just by increasing the number of hidden units (Christiansen and Chater 1999). 20 hidden units were used throughout this paper as it guaranteed training to a desired accuracy level was possible, up to a limit of about 80% accuracy.

[2]As originally studied by Elman (1990), SRNs were trained using a simple backpropagation procedure that did not propagate error signals across previous timesteps. During learning this procedure follows a trajectory that is not guaranteed to follow the error gradient. In contrast, SRBTT follows the error gradient exactly, and is therefore more like to result in successful learning. In informal tests, we discovered that SRNs trained using simple backprop did not perform as well as those trained using SRBPTT. Networks with 20 hidden units were unable to achieve the desired accuracy level within the number of training epochs allotted. Following Rodriguez (2001), we continue to use the term SRN despite the departure from the original learning algorithm used to study the architecture.

[3]Despite all networks achieving the desired accuracy level within 300 epochs, some tasks took longer to train than others. In particular, networks trained to perform forward recall took longer to reach 80% accuracy than those trained to only perform backward recall. In some sense, forward recall may be "harder" than backward recall. In Grüning's (2006) formulations of backward and forward recall presented above in (9) and (10), this could be interpreted as a consequence of the extra counter variable y that needs to be managed in the forward but not backward case. More generally, we might expect forward recall to be more difficult than backward recall as the type of grammars it models are context-sensitive rather than context-free, and the context-sensitive grammars form a more complex superset of the context-free grammars in the Chomsky hierarchy of languages. Bach et al. (1986), however, found that Dutch speakers perform better with increasing numbers of cross-serial dependencies than German speakers perform with increasingly many levels of center embedding. This suggests that, for humans, the forward recall task is

The reason networks were only trained to 80% accuracy was so their errors could be examined. In past comparisons of network and human performance on serial recall tasks, the types of errors networks made were used as a "fingerprint" of the unknown algorithm used by either the trained network or the brain to perform the task (Botvinick and Plaut 2006, Goldberg and Rapp 2008). If two networks have the same fingerprint (make similar errors) on a particular task, that would be evidence that the perform the task in a similar way.

4.1. ***Tasks Learned***

4.1.1. Basic Forward Recall

In the forward recall task, networks were trained to act as queues. Queue-like networks were trained to handle input/output mappings of the form shown in (15).

(15) Sample input/output mapping for a queue SRN. F for "forward" is a cue to read the next symbol at the front of the queue. N indicates that network output is irrelevant at that timestep.

input:	`0101FFFF`
output:	`NNNN0101`

As can be seen from the sample mapping, network operation can be divided into timesteps devoted to encoding a string of symbols (the first half of the total timesteps), and timesteps devoted to recalling the stored string.

Once a network had been trained to 80% accuracy, it was tested on the full set of possible input/output mappings to get a complete picture of the errors it makes. Since we are looking at binary input strings of length 1 through 9, this test set consisted of $\sum_{n=1}^{9} 2^n = 1022$ strings. The error curves shown in figure 3 represent averages over 20 trained networks, and are presented with 95% confidence intervals. The curves are derived from recall errors on input strings of length 5, 6, and 7. The number of possible input strings at shorter lengths is too low to produce reliable curves (e.g. there are only 8 strings of length 3), and network performance breaks down for longer input strings. Each position corresponds to a position in the initial input string (e.g. the first position counts errors made when the first symbol of the input was recalled). The y axis of the curve represents the ratio of errors made in a particular position over the number of possible errors that could have been made. Since there were 2 possible output symbols[1], chance performance is at 50%.

As test string length grows, a u-shaped error curve emerges, as found in various human studies of forward serial recall and by Botvinick and Plaut (2006).[2]

The error curves shown above are derived only from network performance during the recall phase of operation. To test if networks encode strings as predicted in the

easier than the backward recall task. Similarly, Christiansen and Chater (1999) showed that networks trained to perform a task similar to the forward recall task presented here (their task more directly simulated cross-serial agreement between nouns and verbs) were able to achieve an overall lower error level than networks trained to perform a center-embedding task, given the same number of hidden units.

[1]There was also a null output symbol that could be active during encoding, but networks never output it during recall.

[2]Unlike Botvinick and Plaut, Goldberg and Rapp (2008) found only monotonically increasing error curves in their SRN experiments on forward serial recall. They used this observation as support for the claim that SRNs recall strings via a serial chaining mechanism. However, the networks used by Rapp and Goldberg differed substantially from those in this paper and in Botvinick and Plaut. The input units of Rapp and Goldberg's networks encoded the entire input string for every timestep of recall (e.g. 9x26 input units encoding 9 positions, each containing one letter). The network only had to output each input symbol in succession, rather than first memorize the input string. Furthermore, error was induced by the addition of gaussian noise to the network's hidden units after training, rather than by stopping training after a certain accuracy was achieved.

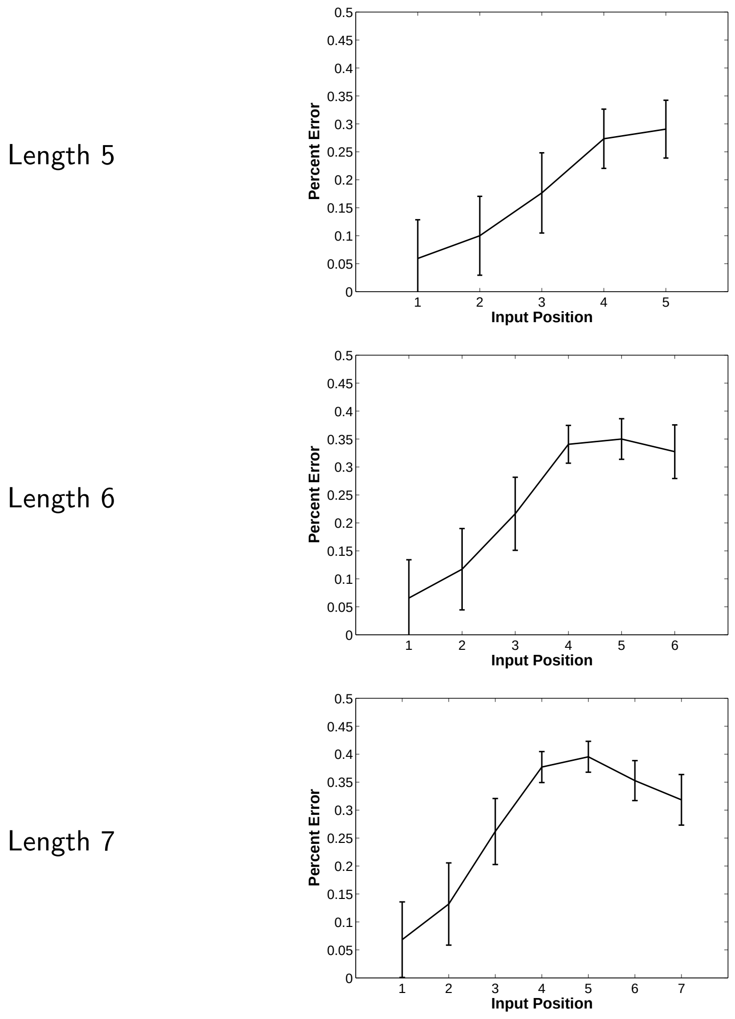

Figure 3. Forward recall error by input string position for strings of length 5, 6, and 7. Curves shown are averaged over 20 networks trained to 80% accuracy, and are presented with 95% confidence intervals. Each position corresponds to a position in the initial input string (e.g. the first position counts errors made when the first symbol of the input was recalled). The y axis of the curve represents the ratio of errors made in a particular position over the number of possible errors that could have been made. As test string length grows, a u-shaped error curve emerges.

equations above, we used the pairwise correlation between hidden unit vectors to determine the similarity of network states at various time steps during the encoding phase.[3] That is, the hidden unit vector at each time step was correlated with hidden unit vectors at all other timesteps for a particular input string. Each measurement is an average over all possible test strings taken from a single network trained to 80% accuracy, but results were nearly identical across multiple simulations. All subsequent correlation measures presented in this paper were also taken from single networks trained to 80% accuracy. Figure 4 shows correlation results for basic forward networks. The color of each square represents the absolute correlation between two positions. The colors are scaled so that a correlation of -1 is represented

[3]Similarly, Servan-Schreiber et al. (1991) used the Euclidean distances between hidden unit vectors to cluster network states. Their analysis indicated that clusters corresponded to states in a finite-state machine their SRNs were simulating.

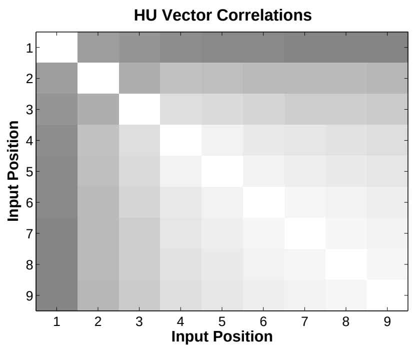

Figure 4. Encoding position correlations for basic forward networks. The hidden unit vector at each time step of encoding was correlated with hidden unit vectors at all other timesteps for a particular input string. Each measurement (square) is an average over all possible test strings. The color of each square represents the absolute correlation between two positions. The colors are scaled so that a correlation of -1 is represented by black, 1 by white, and intermediate values by shades of gray.

by black, 1 is represented by white, and intermediate values are represented by shades of gray. As seen in the figure, basic forward networks show an asymmetry during encoding. An input symbol at a particular position is more similar to the symbols that come after it than the symbols that come before. This is consistent with the forward recall equations seen in (10). Note that in the equations, each subsequent symbol in the input string is placed in the next available *least*-significant position in a binary expansion. Thus, later encoding operations operate in smaller and smaller subspaces (e.g. hundredths then thousandths, etc.), leading to more similar network states during those operations. This process can be visualized by observing the plot on the left of Figure 2. As a string is encoded, we follow a path from the root of the fractal tree upward. At each branch point, the distance between the two outgoing branches becomes smaller and smaller, indicating higher correlation values between network states.

The actual pattern of values predicted by the equations is shown in Figure 5. Since there is only one stack variable, x, and one counter variable y referred to in the equations, rather than a vector of hidden unit values, we used Euclidean distance between (x, y) pairs at different timesteps in lieu of correlations. Each square in the figure represents e^{-d}, where d is the average distance between the two points, in order to make the black to white scale match that used in Figure 4 (i.e. the maximum possible "similarity" of 1 occurs when two points are 0 distance apart). The figure is scaled so that black represents $e^{-d_{max}}$, where d_{max} is the maximum average distance encountered in the simulations, and white represents 1 (or a distance of 0). Figures 4 and 5 are very similar, aside from scaling differences due to the use of correlations or distances. Taking the correlation between the values in the two figures (a measure which ignores linear scaling differences), yields a correlation coefficient of 0.987. This is further evidence that networks implement a version of the equations presented above.

In Rodriguez (2001), network behavior is directly analyzed by plotting the trajectory of each hidden unit and speculating as to how each might contribute to overall network function. Rodriguez performed this analysis on very small networks with just 5 hidden units. In Grüning (2006), a similarly direct analysis was performed, this time using the principal components of the hidden unit activations rather than the hidden unit values themselves. Again, only networks with 5 hidden units were analyzed. Even with small networks, our own informal experiments have

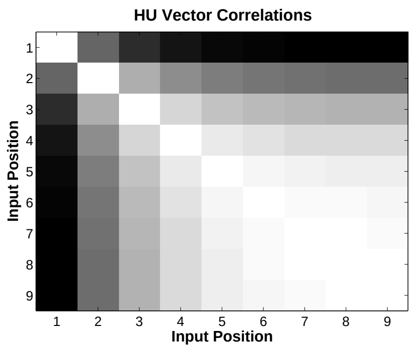

Figure 5. Average distances between points at different encoding positions using idealized forward recall equations. Each square in the figure represents e^{-d}, where d is the average distance between the two points, in order to make the black to white scale match that used in Figure 4 (i.e. the maximum possible "similarity" of 1 occurs when two points are 0 distance apart). The figure is scaled so that black represents $e^{-d_{max}}$, where d_{max} is the maximum average distance encountered in the simulations, and white represents 1 (or a distance of 0).

indicated that the function of individual hidden units or principal components with respect to overall network operation is often not trivial to discover. With larger networks, such direct analysis of network function is entirely infeasible. In contrast, the technique we employ here, the comparison of actual hidden unit correlations to predicted distances between idealized network states, is designed to work well independently of network size. It allows us to gauge if the network is performing as predicted regardless of how many hidden units it has, knowing only the relative sizes of the geometric subspaces used by the network.

4.1.2. Basic Backward Recall

In the backward recall task, networks were trained to act as stacks. Stack-like networks handled mappings of the form shown in (16).

(16) Sample input/output mapping for a stack SRN. *B* for "backward" is a cue to pop off the top symbol on the stack. *N* indicates that network output is irrelevant at that timestep.

input:	`0101BBBB`
target:	`NNNN1010`

Again, network operation can be divided into two phases - encoding (the pushes) and recall (the pops).

Again, once a network had been trained to 80% accuracy, it was tested on the full set of possible input/output mappings ($\sum_{n=1}^{9} 2^n = 1022$) strings. The error curves shown in figure 6 are arranged in the same way as the demonstrative error curves for forward recall networks. That is, a position on one of these curves corresponds to a position in the input string. Note that, as a result of this, unlike in the forward recall case, the *first* position on a curve corresponds to the *last* symbol output.

Unlike forward recall networks, networks trained to perform basic backward recall do not consistently show a u-shaped error curve. Instead, error decreases monotonically with input string position. This is to be expected given the general loss of information over time, since, during backward recall, the first symbols of the input string are recalled last. Interestingly, humans performing the backward recall task show a u-shaped error curve similar to that produced in the forward networks (Li and Lewandowski 1995).

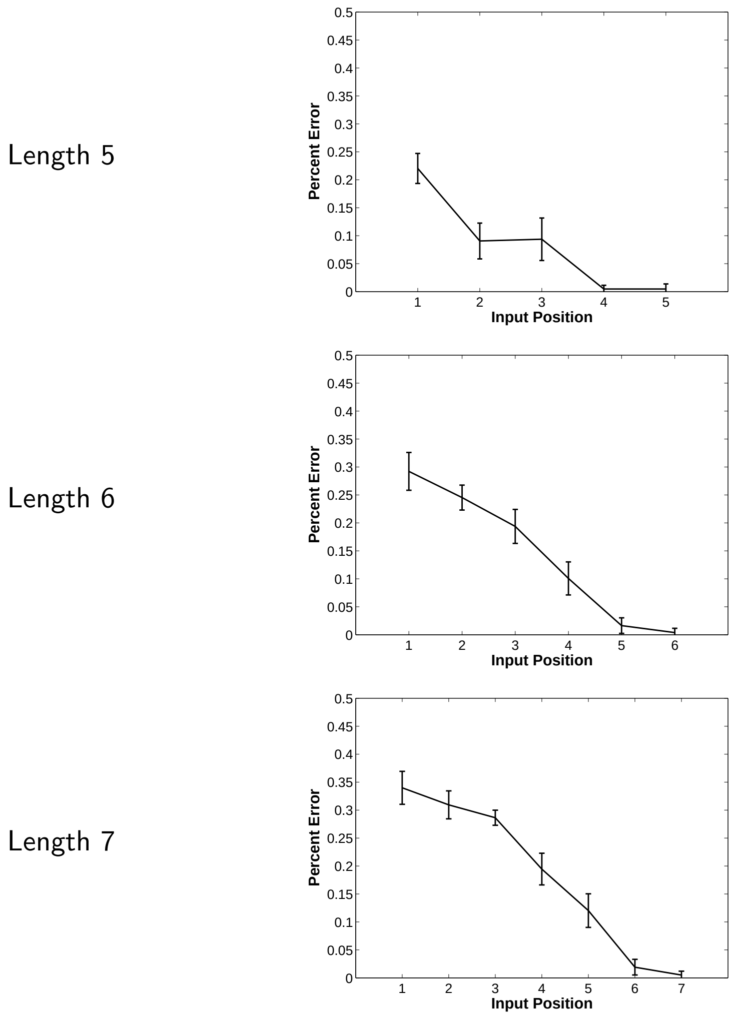

Figure 6. Backward recall error by input string position for strings of length 5, 6, and 7. Note that because this is backward recall, the *first* position on a curve corresponds to the *last* symbol output. Unlike in forward recall networks, error decreases monotonically with input string position.

Again, we looked at the pairwise correlation between hidden unit vectors during encoding. Each measurement is an average over all possible test strings. Figure 7 shows correlation results for basic backward networks. Unlike forward networks, basic backward networks show relatively uniform correlation across all timesteps. Again, this is consistent with the backward recall equations seen in (9). In these equations, each subsequent symbol in the input string is placed in the *most*-significant position in a binary expansion. Thus, all encoding operations operate in roughly the same size subspace, leading to uniform similarity cross network states.

Once again, the correlations show a similar pattern to that predicted by the backward recall equations in (9). The pattern of values predicted by the equations is shown in Figure 8. In the backward case, each cell in the figure represents e^{-d}, where d is the Euclidean distance between stack variable x at different times during encoding. The correlation between the values in Figures 7 and 8 is 0.9423, again suggesting that equations 9 do a good job of approximating network behavior.

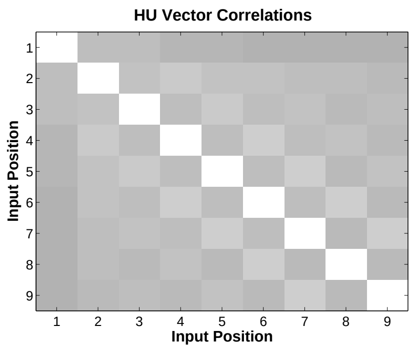

Figure 7. Encoding position correlations for basic backward networks. Values are calculated as in Figure 4.

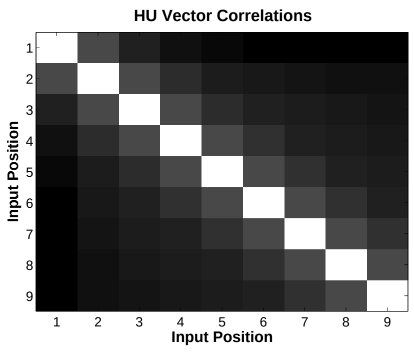

Figure 8. Average distances between points at different encoding positions using idealized backward recall equations. Values are calculated as in Figure 5.

4.1.3. Mixed Recall

Having seen how networks perform as either stacks or queues, we now examine network performance on a combination task that requires reading a stored string from either end. In this task, forward recall and backward recall are mixed between examples. For any given example, a network would need to behave as either a stack or a queue. Using the same parameters as forward and backward networks above, networks were trained to handle the input/output mappings of both types, i.e., like those in (15) and (16). Thus, the mixed task is a union of the basic forward and basic backward tasks.

Since this task includes both basic forward and basic backward input/output mappings, trained networks were tested on a set of $1022 + 1022 = 2044$ strings. Figure 9 shows characteristic error results, once again an average over 20 trained networks. Error for forward mappings, backward mappings, and the full set of mappings is shown.

If the network had implemented forward and backward recall as independent subnetworks, we might expect forward and backward error curves to look like their counterparts generated by single-task networks. As compared to their single-task network counterparts, a u-shape is more prominent in the forward error curves and emerges in the backward error curves where it was not previously found. In the backward case, however, the turning point of the u-shape occurs after the second position in the input string for every string length, even though the first two input string positions are always output last. The turning points of the u-shaped curves for these cases are also very sharp, unlike the gradual curvature seen in the forward

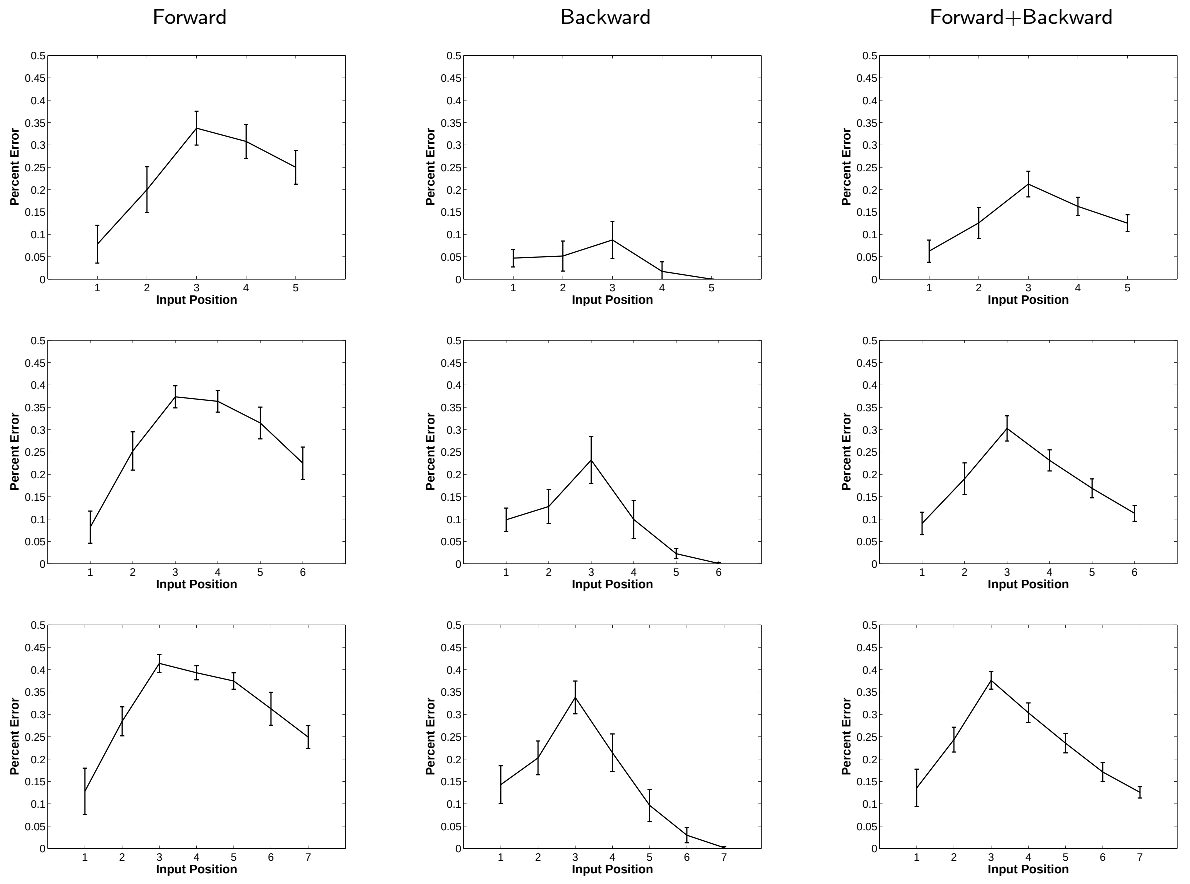

Figure 9. Forward, backward, and combined error by input string position for strings of length 5, 6, and 7 using mixed recall. Forward error curves are derived only from queue-like inputs. Backward error curves are derived only from stack-like inputs. The combined error curve is derived from all examples tested. Forward and backward curves are similar to their single network counterparts, aside from the sharp downturn in error in the first two positions of the backward curves.

case. This suggests that the u-shape seen in the backward cases of the dual-task networks might be an artifact of the networks treating the first two input positions in some special way. Aside from these two positions, error increases monotonically by input position, just as in single-task networks.

Although there are differences, such as the u-shape that emerges in the backward error curve, the error curves are similar to their single network counterparts.

Looking at the pairwise correlation between hidden unit vectors during encoding in Figure 10, we see behavior similar to the basic forward networks. An input symbol at a particular position is more similar to the symbols that come after it than the symbols that come before. This suggests that mixed networks may operate in way that encodes each subsequent input symbol in the next least-significant slot of a binary expansion. However, we also see behavior similar to the basic backward networks - an input symbol at a particular position is equally similar to input symbols at both preceding and following nearby positions, unlike the basic backward networks considered above.

Figure 11 shows the correlation pattern predicted by the independent forward/backward recall equations in (12). There are three variables manipulated in these equations, the stack variable x_1, the queue variable x_2, and the counter y,

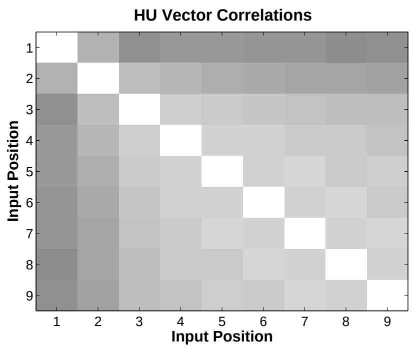

Figure 10. Encoding position correlations for mixed networks. Values are calculated as in Figure 4.

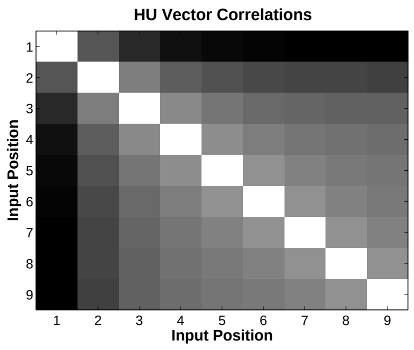

Figure 11. Average distances between points at different encoding positions using idealized forward/backward recall equations with independent data structures. Values are calculated as in Figure 5.

so each cell in the figure represents e^{-d}, where d is the Euclidean distance between (x_1, x_2, y) triplets at different points during encoding. The predicted pattern of correlations is very similar to that derived from actual hidden unit vectors, with a correlation of 0.9783, suggesting that mixed networks are implementing forward and backward recall independently by storing a stack and a queue separately.

For comparison, figure 12 shows the correlation pattern predicted by the forward/backward recall equations using a shared representation and independent pointers in (13). The encoding dynamics used by the non-independent forward/backward recall equations in (14) are identical, and so are also represented by figure 12.[1] There are two variables manipulated, the stack variable x and the pointer y_2, so each cell in the figure represents e^{-d}, where d is the Euclidean distance between (x, y_1, y_2) triplets at different points during encoding. Since y_2 is multiplied by 2 at each time step, its value quickly overwhelms the value of x. Thus, we see very little similarity between different positions. This does not appear to match the behavior of trained networks, and is thus a less likely explanation for their behavior than the independent representation equations in (12). The correlation between Figures 10 and 12 is 0.6987, lower than the 0.9783 obtained from figure 11. So, it seems that mixed networks tend to use independent representa-

[1]The encoding dynamics in (13), with pointers and no data depletion, were described as a possible strategy for forward recall networks in (11). However, the correlation between the network behavior in Figure 4 and the model predictions in Figure 12 is only .4181, suggesting that such encoding dynamics are not used by networks trained to perform forward recall.

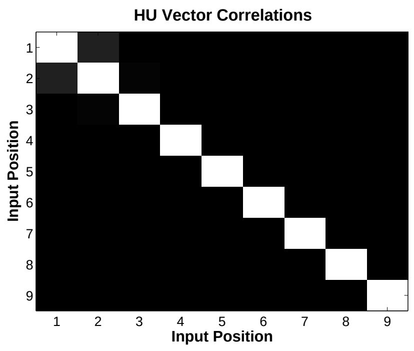

Figure 12. Average distances between points at different encoding positions using idealized forward/backward recall equations with independent pointers. Values are calculated as in Figure 5.

tions, rather than shared representations and potentially independent pointers, to encode strings for both forward and backward recall.

4.2. *Experiments with Trained Networks*

4.2.1. Extended Recall For Mixed Task

By comparing correlations between hidden units to the distances predicted between different network states by our idealized equations, we have provided evidence that mixed networks perform forward and backward recall independently. If such networks maintain two independent data structures for forward and backward recall, or if they use two independent pointers to access a single data structure (less likely to be the case, as shown above), they should be able to recall the entire input string twice - once in the forward direction and once in the backward direction. The entire data structure should be available for both forward and backward recall regardless of which process occurs first.

Note that the mixed task only required networks to output the stored input string once. So, networks trained on this task were tested to see if they could generalize to the extended recall task described above. In particular, networks had to handle mappings of the form shown in (17), where the input string needed to be output fully twice, one time in each recall direction. To facilitate this experiment, the networks examined in this paper were never trained to output a special end symbol at the end of recall, unlike the networks in Botvinick and Plaut (2006) and Goldberg and Rapp (2008). Thus, they were not forced to implement a stopping rule based on a count of the number of recall timesteps passed, and are not prevented from generalizing to the extended recall case in principle.

(17) Sample input/output mapping for a stack SRN. *B* for “backward” is a cue to pop off the top symbol on the stack. *N* indicates that network output is irrelevant at that timestep.

input:	`0101BBBBFFFF`
target:	`NNNN10100101`

Figure 13 show how well networks trained on the regular mixed task generalize to the extended recall task. The solid lines represent the first full recall of an input string, and the dashed lines represent the second full recall.

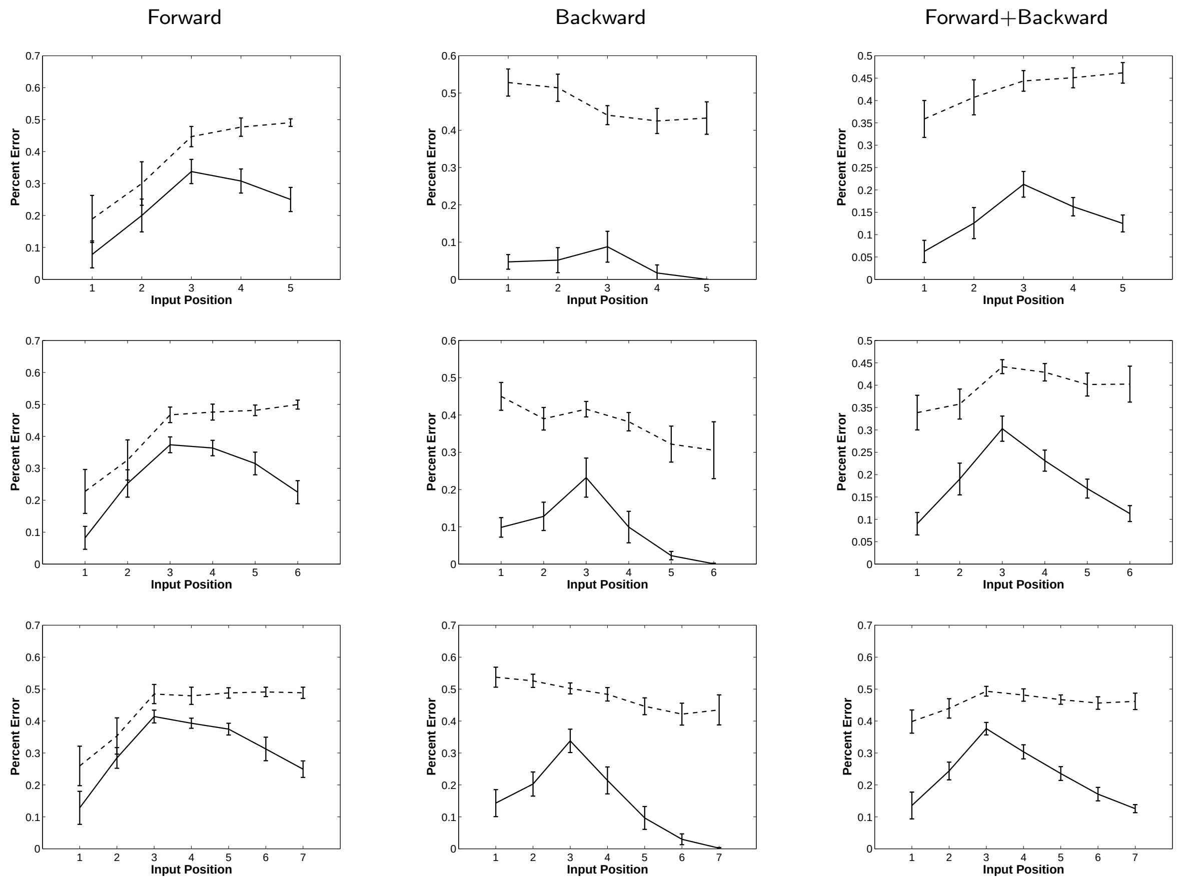

Figure 13. Recall error by input position for strings of length 5, 6, and 7, using a mixed network. The figure shows how well networks trained on the regular mixed task generalize to the extended recall task. Solid lines represent the first full recall of an input string, and the dashed lines represent the second full recall.

Generally, it seems that networks are able to generalize to the extended test set. Overall error during the second full recall is higher than during the first recall, but usually stays below the 50% chance level. In addition, the error curves for the second full recall seem to be slightly flattened and upshifted versions of the first recall curves. These results are consistent with the idea that networks implement forward and backward recall independently, and that the stored input string remains available for recall in one direction after it has been recalled in the other direction. The increased error seen during the second full recall can be accounted for by an overall increase in network error the longer the network is required to operate. Presumably, each time step of recall introduces inaccuracy into the network's internal representations which accumulates over time.

We test if the above analysis is plausible by using a regression model to predict the error curves shown. The predictors used in the regression were the position in the input string of the symbol to be output, the input string length, the number of timesteps passed since the network encoded the symbol to be output, the absolute number of timesteps the network had been running, and whether the output cue was for forward or backward recall. If the model is trained only on the timesteps when a string is recalled for the first time, yet can predict the errors made during the second recall, where the only predictors that differ are the number of timesteps the network has been running for and the number of timesteps passed since it encoded the target symbol, we would have evidence that the increased error during second recall is a result of primarily these factors.

We chose to use a logistic regression model. However, such a model is used to predict quantities that vary between zero and one (usually probabilities). Network error effectively varies between zero and 0.5 (chance). Higher error than 0.5 would indicate that the network is systematically outputting the opposite symbol from the target. In such a case, the network would have memorized the target correctly, and we could recover the symbol simply by flipping the network's output. As shown above, error levels above 0.5 do not arise in practice.

A quantity related to error level is the mutual information between the target symbol a network should output and symbol it actually does output. Mutual information is an information-theoretic quantity that indicates how much the uncertainty about the value of one random variable is reduced by knowing the value of a related random variable[1]. In this case, there are two possible target symbols and two output symbols, so each target can be represented by a bit. At best, knowing the output symbol tells us precisely what the target symbol was, and we learn one bit of information. In the worst case, knowing the output symbol doesn't allow us to predict the value of the target at all. In this case (at the 0.5 error level), we gain zero bits of information. Thus, mutual information varies between zero and one, and can be used as the dependent variable in a logistic regression. We simply treat the mutual information as a probability.[2]

Model predictions are seen as the thin dotted lines in Figure 14. The y-axis in the figures represents one minus the mutual information between the target and input symbols, in order to be comparable to the error curves presented above. The thicker solid and dashed lines represent observed mutual information values derived from the networks used to train the regression model. Solid lines represent first recall, and dashed lines represent second recall. As the model fits a logistic curve, it cannot accurately capture the u-shape present in the training data. Nevertheless, for longer strings, particularly in the backward recall direction where the curves tend to be less u-shaped, the model closely approximates the mutual information derived from actual networks. The model does not match network behavior well for shorter strings, though this may be due to a sparsity of training data (i.e. there are only 16 test strings of length 4). The model was trained only on timesteps from the first recall of each test string (the data used to generate the lower, solid lines in the figure), but does well at capturing the error behavior of the second recall. In particular, it captures how longer network operation (measured in terms of number of timesteps the network has run for, and the number of timesteps passed since the symbol to be output at a particular timestep was recalled) causes error to accumulate over time until network accuracy is reduced to chance levels, from which they cannot recover. That is, it is reasonable to conclude that differences in network accuracy between first and second recall are entirely the result of accumulating noise in network operation over time, rather than a fundamental inability of the network to perform recall a second time. This supports, then, the hypothesis that the network is making use of independent representations for carrying out forward and backward recall, as in the equations in (12).

[1]For two random variables X and Y, the mutual information between them is defined as $I(X;Y) = H(X) - H(X|Y)$. The mutual information is the entropy (uncertainty) in X minus the conditional entropy in X having seen Y. For a full discussion of mutual information, see Cover and Thomas (2006)

[2]For each point on a curve to be regressed against, the number of possible errors that point was divided into "counts" in proportion the observed mutual information between the target and output symbols at that points, and one minus the observed mutual information. These counts were then input to the logistic regression.

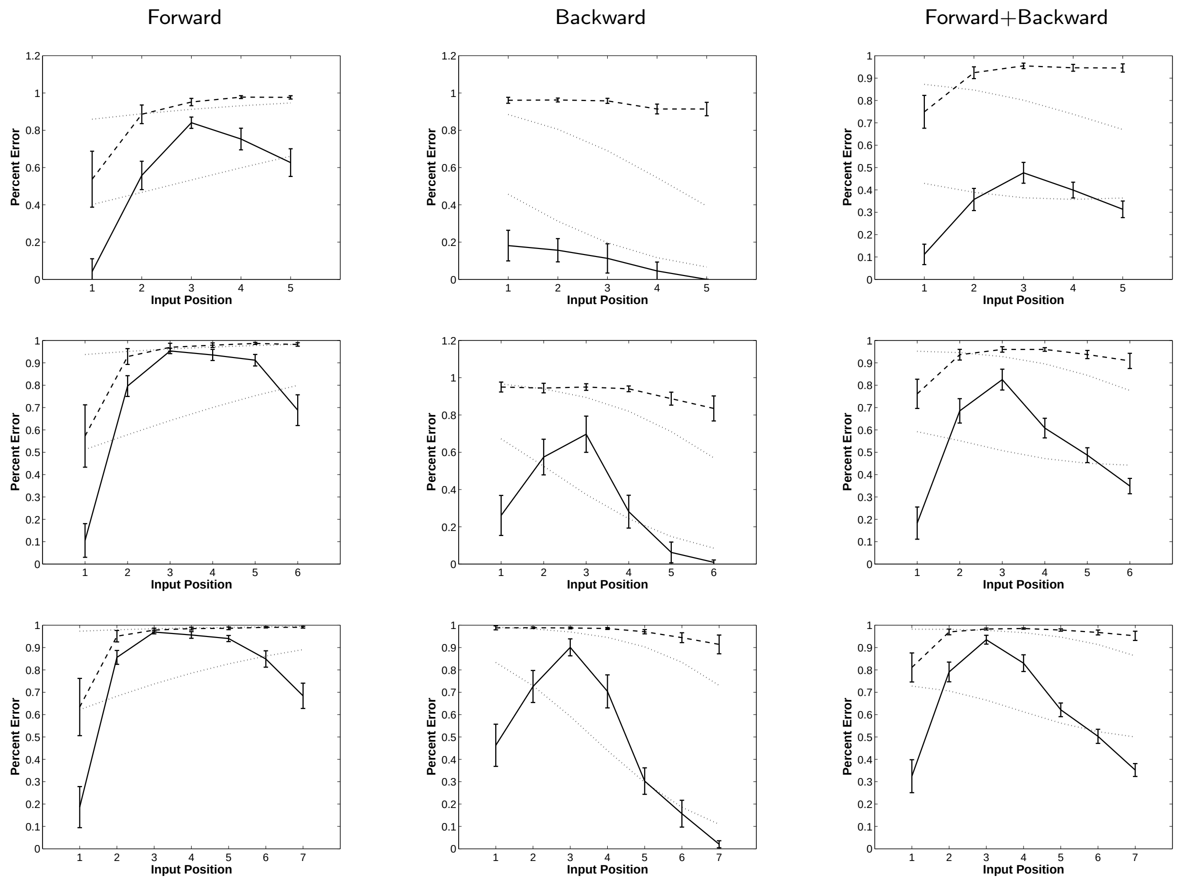

Figure 14. 1-Mutual Information for strings of length 5, 6, and 7 in mixed networks. Dark solid and dashed lines represent mutual information computed from network outputs during first and second recalls respectively. These values are used to train the regression model. The thin dotted lines represent logistic model predictions for first and second recalls.

4.2.2. Correlations Between Memory States

One of the questions raised in the section describing idealized networks above is whether networks actively deplete a stored sequence at recall, altering their stored data structure at each recall timestep, or if they perform recall using a shifting pointer into a fixed data structure. If the former is true (networks actively deplete a data structure during recall) then, given two different input strings with a common substring, a network recalling these two strings should be in an identical numerical state at the point of recall when only the common substring remains in the data structure. In the latter case (networks perform recall using only pointers) the network would never reach a numerically identical state, as the stored data structure remains fixed throughout recall, and always contains two different input strings despite their common substring.

To see which of these situations was more likely to be true, the average pairwise correlation between network states that represented a timestep when input strings with a common substring were recalled up to the point when only their common substring remained to be output. Data was derived from single networks trained to 80% accuracy.

For basic forward networks, strings of the form *XXX111* were considered, where

XXX represents and arbitrary prefix. Pairwise correlations were taken between hidden unit vectors at the fourth timestep of recall, when only the substring *11* was left to be recalled. Using this procedure, the same position (fourth) in the input string is compared every time (effectively, the pointer into the stored sequence should be the same), and output symbol is always the same, removing two potential confounds. The average pairwise correlation between this set of timesteps was 0.832. Similarly, for basic backward networks, strings of the form *111XXX* were considered. Once again, pairwise correlations were taken at the fourth timestep of recall, when only the substring *11* remained on the stack. Average pairwise correlation in this case was 0.837. For mixed networks, both of the above string types were considered, with two sets of correlations corresponding to forward and backward input mappings. The average forward correlation was 0.977. The average backward correlation was 0.959.

All pairwise correlations indicated a high level of similarity between timesteps representing networks with different recall histories (and different initially stored strings), but identical sequences of symbols remaining to be output. This result suggests that trained networks perform recall by depleting (altering) a stored data structure, rather than altering a pointer into a fixed data structure. This is consistent with the independent idealized equations in (12), a result also suggested by the experiments from extended recall.

An alternative possibility is that hidden unit vectors tend to be highly correlated in general, rendering the above correlations uninformative. To account for this, we present hidden unit correlations among the first time-steps of recall for the strings used above. When recall first begins, the contents of the data structure stored by the network are different for each string, and we would expect lower correlations between network states. For forward networks, network states at the onset of recall had a correlation of 0.6180. For backward networks, the correlation was 0.5477. For mixed networks, the correlation for forward recall was 0.7764, and the correlation for backward recall was 0.5259. In all cases, these correlations are lower than those found at time-steps when the remaining contents of the networks' stored data structures match.

5. Conclusion

Through an examination of the correlations between hidden unit vectors, we were able to provide further evidence that SRN's do indeed store strings in a data structure similar to a binary expansion. Depending on how each network implements storage either the beginning or end of the input string to be stored is relegated to a smaller and smaller subspace of hidden unit space (i.e. less significant location in the expansion).

Furthermore, we provided evidence that the process of recalling a stored string by an SRN depletes the stored data structure, much like a pop or a read would do for a symbolic implementation of a stack or a queue. Trained networks do not seem to operate like random access arrays, where a pointer into a data structure can retrieve data without altering the contents of the data structure. However, it also seems that multi-task networks implement independent stacks and queues independently, so that each can be depleted as needed. This allows such networks to fully recall a string a second time in the forward or backward direction after it has already been recalled once in the other direction.

To the extent that SRNs serve as approximate models of human linguistic performance, this behavior has implications for the possible range of recursive processes found in human languages. In particular, it may be possible to have both center-embedded and cross-serial dependencies between the same two items (e.g. a noun phrase and the verb it agrees with) in a single sentence. Once one type of dependency has been expressed, the items would be removed from one internal data structure but would still be present in another, and would thus be available to the grammar. Similarly, we can make predictions about human memory performance. If recall in one direction depletes an underlying representation of stored data, but leaves alone a second independent version of the data, we would expect subjects to be able to perform a second task with an input string after it has already been recalled once. For example, backward recall of a string should be possible (although perhaps more difficult due to memory degradation over time) to perform after the string has been recalled in the forward direction. A confounding factor, however, is that subjects would have access to their own outputs during recall. To the extent that these are accurate, it may be that subjects "refill" their data structure using this information. This would mitigate the effect of depletion during recall. SRNs, as presented in this paper, do not have access to their own outputs.[1]

Finally, the results presented here have more general implications for the study of recurrent neural networks. They suggest that SRNs trained on multiple tasks will perform as multiple individual SRNs, each trained on one task. In particular, while a multi-task network may learn to perform several tasks about as well as single-task networks, this may come at the cost of redundancy. Common aspects of all tasks (e.g. the input strings given to SRNs in this paper) will not be collapsed into a single representation, forcing hidden units to represent more information.

Acknowledgements

We would like to thank Don Mathis, Paul Smolensky and the SRN-Language Research Group at Johns Hopkins for helpful comments and discussion. We are also grateful to the NSF for their financial support of this project in the form of grants SBR-0446929 and SBR-0549379.

References

Li, S., and Lewandowski, S. (1995), "Forward and backward recall: Different retrieval processes," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21, 837–847.

Bresnan, J., Kaplan, R.M., Peters, S., and Zaenen, A. (1982), "Cross-serial dependencies in Dutch," *Linguistic Inquiry*, 13(4), 613–635.

Huybregts, M. (1976), "Overlapping dependencies in Dutch," Utrecht working papers in linguistics, University of Utrecht, Utrecht.

Stabler, E. (2004), "Varieties of crossing dependencies: Structure dependence and mild context sensitivity," *Cognitive Science*, 28(5), 699–720.

Page, M.P., and Norris, D. (1998), "The Primacy Model: A New Model of Immediate Serial Recall," *Psychological Review*, 105(4), 761–781.

Henson, R. (1998), "Short-term Memory for Serial Order: The Start-End Model," *Cognitive Psychology*, 36, 73–137.

Chomsky, N. (1959), "On certain formal properties of grammars," *Information and Control*, 2, 137–167.

Joshi, A.K. (1985), "How much context-sensitivity is required to provide reasonable structural descriptions: tree adjoining grammars," in *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives* eds. D. Dowty, L. Karttunen and A. Zwicky, Cambridge: Cambridge University Press, pp. 206–250.

Joshi, A.K. (1990), "Processing Crossed and Nested Dependencies: an Automaton Perspective on the Psycholinguistic Results," *Language and Cognitive Processes*, 5(1), 1–27.

Joshi, A.K., and Schabes, Y. (1997), "Tree-adjoining grammars," in *Handbook of Formal Languages, Volume 3: Beyond Words* eds. G. Rozenberg and A. Salomaa, New York: Springer, pp. 69–124.

[1]Such refilling can be modeled with network architectures that differ from SRNs. Botvinick and Plaut (2006) trained networks with recurrent connections from output units back to the hidden units. Similarly, Christiansen and Chater (1999) fed network output back into the network as input on the next timestep as part of a sentence generation task.

Miller, G., and Chomsky, N. (1963), "Finitary Models of Language Users," in *Handbook of Mathematical Psychology, Volume 2* eds. R.D. Luce, R.R. Bush and E. Galanter, New York: John Wiley and Sons, pp. 419–491.

Christiansen, M.H., and Chater, N. (1999), "Toward a Connectionist Model of Recursion in Human Linguistic Performance," *Cognitive Science*, 23, 157–205.

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

Wiles, J., and Elman, J. (1995), "Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks," in *Proceedings of the Seventeenth Annual Meeting of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 482–487.

Tabor, W. (2000), "Fractal Encoding of Context Free Grammars in Connectionist Networks," *Expert Systems*, 17(1), 41–56.

Rodriguez, P. (2001), "Simple Recurrent Networks Learn Context-Free and Context-Sensitive Languages by Counting," *Neural Computation*, 13(9), 2093–2118.

Servan-Schreiber, D., Cleeremans, A., and McClelland, J. (1991), "Graded State Machines: The Representation of Temporal Contingencies in Simple Recurrent Networks," *Machine Learning*, 7, 161–193.

Sigelmann, H.T., and Sontag, E.D. (1995), "On the computational power of neural nets," *J. Comput. System. Sci.*, 50(1), 132–150.

Grüning, A. (2006), "Stack-Like and Queue-Like Dynamics in Recurrent Neural Networks," *Connection Science*, 18(1), 23–42.

Badecker, W., and Frank, R. (2008), "SRNs and the learning of grammatical agreement: The effect of target-source order and class overlap," in *Proceedings of the 14th Annual Conference on Architectures and Mechanisms for Language Processing (AMLaP)*, Cambridge University.

Frank, R., Mathis, D., and Badecker, W. (2008), "The Acquisition of Anaphora by Simple Recurrent Networks." Manuscript, Johns Hopkins University.

Botvinick, M.M., and Plaut, D.C. (2006), "Short-term memory for serial order: A recurrent neural network model," *Psychological Review*, 113(2), 201–233.

Goldberg, A., and Rapp, B. (2008), "Is compound chaining the serial order mechanism of spelling? A simple recurrent network investigation," *Cognitive Neuropsychology*, 25(2), 218–255.

McCloskey, M. (1991), "Networks and theories: The place of connectionism in cognitive science," *Psychological Science*, 2, 387–395.

Bodén, M., and Blair, A. (2003), "Learning the Dynamics of Embedded Clauses," *Applied Intelligence*, 19, 51–63.

Bach, E., Brown, C., and Marslen-Wilson, W. (1986), "Crossed and nested dependencies in German and Dutch: A Psycholinguistic Study," *Language and Cognitive Processes*, 1(249-262).