Networks which learn to store variable-length sequences in a fixed set of unit activations

Tony A. Plate
Dept of Chemical Engineering
University of British Columbia
2216 Main Mall
Vancouver BC Canada V6T 1Z4
tap@epi.terryfox.ubc.ca

22 March 1995

Abstract

A method for storing sequences of varying lengths in a fixed-width vector is described. The method is implemented, in an adaptive form, in a recurrent network which learns to generate sequences. The performance of this network is compared with that of a more conventional recurrent network on the same task.

1 Introduction

Sequence processing in recurrent networks often requires the storage of information about variable length sequences in the activations of a fixed-size group of units. However, there have been few studies of how this can be accomplished. In one paper on the topic, Simard and Le Cun [1992a] reported difficulties with teaching a recurrent network to generate a variety of trajectories through a continuous space. As well as being interesting for its own sake, the problem of storing sequences in a fixed-size vector of activations is interesting for several other reasons:

- the study of it might give some insight into learning difficulties in recurrent networks for sequence classification. Servan-Schreiber et al [1991] and Maskara and Noetzel [1992] have suggested that slow learning or inability to learn in recurrent networks used for sequence classification is at least in part due to the difficulty of preserving sufficient context in the hidden units.
- a solution could be a useful pre-processing technique for presenting sequential information to a neural network.

In this paper, I investigate the storage of sequences in a recurrent net vector using the task of sequence generation. I use this task rather than the more common sequence classification task because it is a way of dealing directly with issues of sequence representation, without distractions such as problems of recognizing languages. I describe a conventional recurrent network which can learn representations for sequences of variable length, and compare it to a recurrent network which implements an adaptive version of a convolution-based sequence storage technique. The latter network is simpler in that the recurrence function has far fewer parameters.

2 Associative memory for sequences

A simple method for storing sequences in a fixed-size memory is to use an associative memory to associate elements of the sequence with points along a predetermined trajectory. This is akin to the memory aid known as the method of loci, which instructs us to remember a list of items by associating each term with a distinctive location along a familiar path. When used with an associative memory, I call this technique "trajectory association". Although this technique can be used with any style of associative memory, in this paper I only consider using it with an associative memory based on circular convolution. Before describing trajectory association, I give a brief overview of circular-convolution associative memory (for more details, see Plate [1995] or [1994]).

2.1 Circular-convolution associative memory

The convolution operation is the associative memory operator in holographic memory. Its role is analogous to that of the outer-product in matrix memories, e.g., the Hopfield net [Hopfield 1982]. For associative memories, circular (wrapped) convolution has the advantage that vectors storing associations are of the same dimensionality as the vectors being associated. Circular convolution can be viewed as a vector multiplication operator which maps pairs of vectors to a vector (just as matrix multiplication maps pairs of matrices to a matrix). The symbol for circular convolution is $\circledast$ and it is defined as

$$\mathbf{z} = \mathbf{x} \circledast \mathbf{y} \ \text{ such that } \ z_j = \sum_{k=0}^{n-1} y_k x_{j-k},$$

where $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are vectors of dimension n, x_i etc. are their elements, and subscripts are modulo-n (so that $x_{-2} \equiv x_{n-2}$). Circular convolution can be computed in $O(n \log n)$ using Fast Fourier Transforms (FFTs). Algebraically, convolution behaves like scalar multiplication: it is commutative, associative, and distributes over addition. The identity vector for convolution ($\mathbf{I}$) is the "impulse" vector: its zero'th element is 1 and all other elements are zero. Most vectors have an inverse under convolution, i.e., for most vectors $\mathbf{x}$ there exists a unique vector $\mathbf{y}$ ($=\mathbf{x}^{-1}$) such that $\mathbf{x} \circledast \mathbf{y} = \mathbf{I}$. For vectors with identically and independently distributed zero mean elements and an expected Euclidean length of 1 there is a numerically stable and simply derived approximate inverse.[1] The approximate inverse of $\mathbf{x}$ is denoted by $\mathbf{x}^*$ and is defined by the relation $(x^*)_j = x_{n-j}$. Vectors for which the exact inverse is equal to the approximate inverse are especially useful in trajectory association. I call these "unitary" vectors, they are the vectors whose frequency components (i.e., the elements of their discrete Fourier transforms) all have a magnitude equal to one.

A set of pairwise associations is stored as the sum of the circular convolutions of the pairs of vectors. For example, the pairs **a**&**b**, **c**&**d**, and **e**&**f** are stored as $\mathbf{a} \circledast \mathbf{b} + \mathbf{c} \circledast \mathbf{d} + \mathbf{e} \circledast \mathbf{f}$. The result can be decoded by convolving with the exact inverse or approximate inverse, though the latter generally gives more stable results. For example, we can find out what is associated with $\mathbf{c}$ in the above by convolving with $\mathbf{c}^*$:

$$\begin{aligned}
(\mathbf{a} \circledast \mathbf{b} + \mathbf{c} \circledast \mathbf{d} + \mathbf{e} \circledast \mathbf{f}) \circledast \mathbf{c}^* &= \mathbf{a} \circledast \mathbf{b} \circledast \mathbf{c}^* + \mathbf{c} \circledast \mathbf{d} \circledast \mathbf{c}^* + \mathbf{e} \circledast \mathbf{f} \circledast \mathbf{c}^* \\
&= \mathit{noise} + \mathbf{c} \circledast \mathbf{c}^* \circledast \mathbf{d} + \mathit{noise} \\
&\approx \mathbf{I} \circledast \mathbf{d} = \mathbf{d}
\end{aligned}$$

[1] Circular convolution with the approximate inverse of $\mathbf{x}$ is equivalent to correlation with $\mathbf{x}$.

The magnitude of the noise can be quantified (see Plate [1994]) and decreases with increasing vector dimension.

Circular convolution is a linear operation, and can be expressed as a matrix-vector multiplication:

$$\mathbf{a} \circledast \mathbf{b} = M_a \mathbf{b}$$

where M_a is the matrix corresponding to convolution by $\mathbf{a}$ and has elements $m_{a_{ij}} = a_{i-j}$ (the subscripts on $\mathbf{a}$ are interpreted modulo n).[2] Since the mapping computed by the connections between two layers in a feedforward network is a matrix-vector multiplication, it is possible for such a mapping to implement convolution by a fixed vector (if the unit activations are linear).

The algebra carries through with matrix multiplication: $M_{a\circledast b} = M_a M_b$. Transposition corresponds to the approximate inverse, i.e., $M_{a^*} = M_a^T$. Also, if $\mathbf{a}$ is a unitary vector, then M_a is a real unitary matrix (i.e., an orthonormal matrix). The eigenvalues of M_a are the individual (complex valued) elements of the Fourier transform of $\mathbf{a}$. The corresponding eigenvectors are the inverse transforms of the frequency components, i.e., $[1, 0, 0, \ldots]$, $[0, 1, 0, \ldots]$, etc, in the frequency domain. Thus, all the eigenvalues of the matrices corresponding to convolution with a unitary vectors have a magnitude equal to one.

Holographic Reduced Representations [Plate 1995; Plate 1994] use circular convolution for associating elements of a structure in a way that can embody hierarchical structure. The key property of circular convolution that makes it useful for representing hierarchical structure is that the circular convolution of two vectors is another vector of the same dimension, which can be used in further associations.

Among associative memories, holographic memories have been regarded as inferior because they produce very noisy results and have poor error correcting properties. However, the noisy results can be cleaned up with conventional error correcting associative memories. This gives the best of both worlds – the ability to represent sequential and recursive structure and clean output vectors.

2.2 Trajectory association

In the trajectory-association method for storing sequences, elements of the sequence and loci (points) on the trajectory are all represented by n-dimensional vectors. For example, the sequence "abc" can be stored as:

$$\mathbf{s}_{abc} = \mathbf{p}_0 \circledast \mathbf{a} + \mathbf{p}_1 \circledast \mathbf{b} + \mathbf{p}_2 \circledast \mathbf{c},$$

where the $\mathbf{p}_i$ are vectors representing the loci, which should not be similar to each other, and $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are the vectors representing the items. This representation of a sequence can be decoded by convolving it with the approximate inverses of the $\mathbf{p}_i$ in the appropriate order.

An easy way to generate a suitable sequence of loci is to use convolutive powers of a random unitary vector $\mathbf{k}$, i.e., $\mathbf{p}_i = \mathbf{k}^i$. Convolutive powers are defined in the obvious way: $\mathbf{k}^0$ is the identity vector and $\mathbf{k}^{i+1} = \mathbf{k}^i \circledast \mathbf{k}$. Restricting $\mathbf{k}$ to be unitary ensures that $\mathbf{k}$ does not blow up or disappear when raised to high powers, because, for any unitary vector $\mathbf{k}$, and for all real t, $||\mathbf{k}^t|| = 1$;

[2] Matrices with this regular pattern of repeated values are known as "circulant matrices" [Davis 1979].

Direct		Recursive		Expansion			Result
$\mathbf{k}^0 \circledast \mathbf{s}_{abc}$	$=$	$\mathbf{s}_{abc}$	$=$	$\mathbf{a} \quad +\mathbf{b} \circledast \mathbf{k} \quad +\mathbf{c} \circledast \mathbf{k}^2$		$\approx$	$\mathbf{a}$
$\mathbf{k}^{-1} \circledast \mathbf{s}_{abc}$	$=$	$\mathbf{k}^{-1} \circledast \mathbf{s}_{abc}$	$=$	$\mathbf{a} \circledast \mathbf{k}^{-1} + \mathbf{b} \quad +\mathbf{c} \circledast \mathbf{k}$		$\approx$	$\mathbf{b}$
$\mathbf{k}^{-2} \circledast \mathbf{s}_{abc}$	$=$	$\mathbf{k}^{-1} \circledast (\mathbf{k}^{-1} \circledast \mathbf{s}_{abc})$	$=$	$\mathbf{a} \circledast \mathbf{k}^{-2} + \mathbf{b} \circledast \mathbf{k}^{-1} + \mathbf{c}$		$\approx$	$\mathbf{c}$

Table 1: The direct and recursive methods of decoding a trajectory-associated sequence.

Using convolutive powers for the loci, the trajectory-association representation for the sequence “abc” is:

$$\mathbf{s}_{abc} = \mathbf{a} + \mathbf{b} \circledast \mathbf{k} + \mathbf{c} \circledast \mathbf{k}^2.$$

2.3 Decoding trajectory-associated sequences

Trajectory-associated sequences can be decoded by repeatedly convolving with the inverse of the vector that generated the encoding loci. As with all convolution-based associations, the results are noisy and must be cleaned up. This requires all possible items to be stored in the clean-up memory.

There are two exactly equivalent methods for decoding a trajectory-associated sequence, which are shown in Table 1. Since the approximate inverses of the loci are equal to their exact inverses (i.e., $(\mathbf{k}*)^i = \mathbf{k}^{-i}$), I use negative powers to indicate the inverses of the loci. In the direct access method, the inverses of each loci are used to probe the sequence representation:

$$\mathbf{r}_i = \mathbf{k}^{-i} \circledast \mathbf{s},$$

where $\mathbf{r}_i$ is the decoded ith element of the sequence represented by $\mathbf{s}$. In the recursive access method, the sequence representation is repeatedly convolved with $\mathbf{k}^{-1}$, which steps through the elements of the sequence:

$$\mathbf{r}_0 = \mathbf{s} \quad \text{and} \quad \mathbf{r}_i = \mathbf{k}^{-1} \circledast \mathbf{r}_{i-1}$$

These two methods are equivalent because convolution is associative.

For example, to retrieve the third element of the sequence $\mathbf{s}_{abc}$ we convolve twice with $\mathbf{k}^{-1}$, and the result expands to $\mathbf{a} \circledast \mathbf{k}^{-2} + \mathbf{b} \circledast \mathbf{k}^{-1} + \mathbf{c}$. The first two terms are unlikely to be correlated with any items in the clean-up memory, so the most similar item will probably be $\mathbf{c}$.

The similarity of $\mathbf{x} \circledast \mathbf{k}^i \circledast \mathbf{k}^{-j}$ and $\mathbf{x}$ depends on the similarity of $\mathbf{k}^i \circledast \mathbf{k}^{-j}$ and $\mathbf{I}$, which is the same as the similarity of $\mathbf{k}^i$ and $\mathbf{k}^j$. If i and j differ, the expected value of this similarity is $2/n$, and the variance decreases with as n increases. This means that the probability of correct decoding increases with n.

Raising a unitary vector to some power t corresponds to multiplying the phase angles of its frequency components by t.[3] Thus, a unitary vector raised to any power is still a unitary vector. The similarity of two unitary vectors depends on how well their corresponding phase angles match – the dot-product is proportional to the sum of the cosines of the differences of the angles:

$$\mathbf{x} \cdot \mathbf{y} = \frac{1}{n} \sum_j \cos(\theta_j(\mathbf{x}) - \theta_j(\mathbf{y})),$$

[3]Fractional convolutive powers are easily defined in this way.

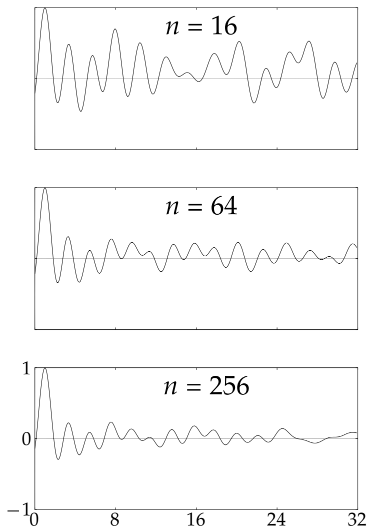

Figure 1: Example trajectory similarities. Each plot shows the similarity (vertical axis) of $\mathbf{k}$ to $\mathbf{k}^t$, for t = 0 to 32 (horizontal axis), for randomly selected unitary vectors $\mathbf{k}$ of dimension n. All plots are to the same scale.

where $\theta_j(\mathbf{x})$ is the phase angle of the jth (frequency) component of the discrete Fourier transform of $\mathbf{x}$.[4] Since the phase angles are randomly distributed in $(-\pi, \pi]$, different integer powers of the same unitary vector have low expected similarity. Of course, for unitary $\mathbf{k}$ with phase angles that are rational fractions of π, there is some t such that $\mathbf{k}^t = \mathbf{k}$. However, this t is almost always very large. Figure 1 shows some plots of the similarity of $\mathbf{k}^t$ to $\mathbf{k}$, for unitary vectors of various dimensions. The first peak in each graph is for $t = 1$ (because $\mathbf{k}^1 = \mathbf{k}$). For large n, randomly chosen $\mathbf{k}$, and $t >> 2$, the expected similarity of $\mathbf{k} \cdot \mathbf{k}^t$ is $2/n$, and the variance decreases as n increases. Note that for t between 0 and 2, $\mathbf{k}$ and $\mathbf{k}^t$ become more similar as t becomes closer to 1. Also, it is easy to show that the similarity of two vectors is invariant under translation in power: $\mathbf{k} \cdot \mathbf{k}^t = \mathbf{k}^\alpha \cdot \mathbf{k}^{\alpha+t}$ for all α. This means, for example, that the similarity of $\mathbf{k}^6$ and $\mathbf{k}^8$ is the same as the similarity of $\mathbf{k}$ and $\mathbf{k}^3$.

The waveforms in Figure 1 are actually the superpositions of $n/2 - 1$ equally weighted sinewaves, and a constant $2/n$.[5] Each sinewave is generated by one frequency component of $\mathbf{k}$, and its period is $2\pi/\alpha$, where α is the phase angle (in $(-\pi, \pi]$).

2.4 Capacity of trajectory-association

In other papers I have given calculations for the capacity of circular-convolution associative memory [Plate 1994; Plate 1995]. In these calculations I assume that the elements of all vectors (dimension n) are chosen randomly from a Gaussian distribution with mean zero and variance $1/n$ (giving an expected Euclidean length of 1.0). Even for short sequences,

[4] This identity is easily proved by substituting $(\mathbf{z} - \mathbf{y})$ for $\mathbf{x}$ in the discrete version of Parseval's theorem, which states that $\sum_k x_k^2 = \frac{1}{n}\sum_k |f_k(\mathbf{x})|^2$, where $f_k(\mathbf{x})$ is the k component of the discrete Fourier transform f $\mathbf{x}$.

[5] The constant of $2/n$ comes from the two frequency components which have a phase angle of zero for real vectors.

quite high vector dimensions are required to ensure a low probability of error in decoding. For example, with 512 element vectors and 1000 items in the clean-up memory, 4 pairs can be stored with about a 1% chance of an error in decoding. The scaling is nearly linear in n – with 1024 element vectors 8 pairs can be stored with about a 1% chance of error. This works out to an information capacity of about 0.16 bits per element. The elements are floating-point numbers, but high precision is not required.

These capacity calculations are roughly applicable to trajectory-association – the number of pairs corresponds to the length of sequences. They slightly underestimate its capacity because of the restriction that the encoding loci are unitary vectors, which results in lower decoding noise. Nonetheless this figure provides a useful benchmark against which to compare the capacity of the adaptive networks described in this chapter, which, on account of the training, should have a higher storage capacity.

2.5 Detecting the end of the sequence

In the above, I have ignored the problem of detecting the end of a sequence. There are two simple ways to do this: (1) use a special end-of-sequence symbol (so that "abc" is stored as $\mathbf{a} + \mathbf{b} \circledast \mathbf{k} + \mathbf{c} \circledast \mathbf{k}^2 + \mathbf{E} \circledast \mathbf{k}^3$, where $\mathbf{E}$ is the end-of-sequence symbol), or (2) use a threshold on the similarity of the decoded sequence element to decide whether the end of the sequence has been reached. Of these, the first method is more reliable.

3 Encoding and decoding systems for trajectory-associated sequences

Figure 2 shows the obvious way in which an encoding subsystem and a decoding subsystem can be composed to form a system which can store and recall sequences. In a system which uses trajectory-association to form distributed codes for sequences, there are at least two opportunities for learning representations. We can learn distributed representations for the individual items (à la Hinton [1986]), and we can fine-tune the distributed code for a sequence. The latter can be useful if errors occur during recall of the sequence – often we can adjust the code so that it is read out correctly.

If the error surfaces of the decoder are simple enough, the idea of adjusting the code for the sequence so that it is read out correctly can be extended to finding the code for a sequence from an arbitrary starting point. To do this, we instantiate the code units with some starting vector (e.g., a random vector), run the decoder to see what sequence it produces, compare that to the sequence we want to produce, backpropagate the errors, and adjust the code units to better produce the desired sequence. We repeat this until the code units contain the vector which produces the desired sequence.

In this chapter I only consider the decoding system, and investigate to what extent gradient descent methods can learn good representations for items and find codes for producing sequences of items. In some ways, this is the opposite of how recurrent networks are usually used to process sequences, in that the sequence is the output rather than the input. The reason I do this is that I wish to focus on representations for sequences. Decoding the sequence representation is a convenient way of ensuring that all the sequential information is present. Furthermore, at this preliminary stage I wish to avoid the complexity of constructing both encoding and decoding systems.

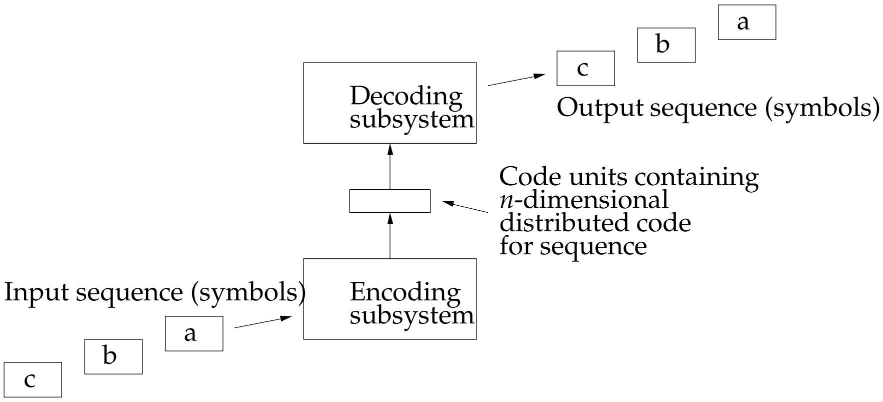

Figure 2: Subsystems to encode a sequence of symbols into a trajectory-associated code, and decode the code back into a sequence of symbols. The encoding and decoding subsystems must store internally the n-dimensional vectors which represent the symbols.

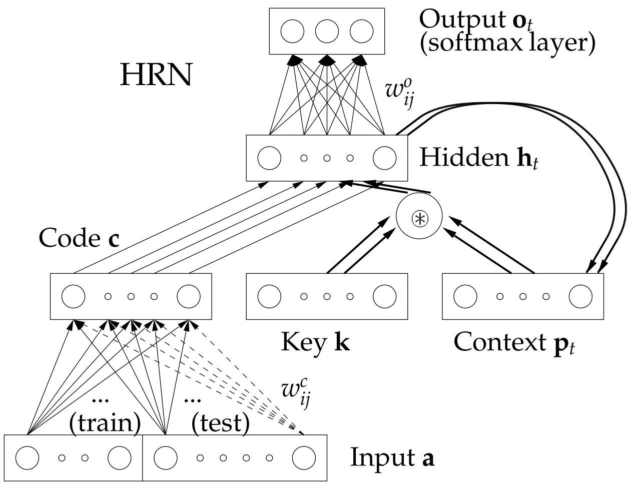

Figure 3: A Holographic recurrent network (HRN). The backwards curved arrows denote a copy of activations to the next timestep. The hidden, code, context, and key layers all have the same number of units. The left block of input units is used during training, the right block during testing.

4 Trajectory-association decoding in a recurrent network

It is straightforward to build the trajectory-association decoding operations (Section 2.3) into a recurrent network. I call the resulting network a *Holographic recurrent network* (HRN).

4.1 Architecture

The HRN used in the experiments described here is shown in Figure 3. The clean-up memory is easily implemented with a "softmax" output layer (See Section 4.2). The output layer does not actually hold distributed representations for symbols, instead it uses a local representation. The softmax function ensures that the total activity in the output layer sums to 1, so that the output activities can be interpreted as probabilities. The output unit with maximum activation will be the one whose incoming weights best match the activations on the hidden units (the incoming weights on the outputs are the distributed representations of symbols). Using a local output representation and the softmax function allows us to take derivatives of the error function. If we wanted to have distributed representations for symbols on the output, we could add an extra output layer and the weights to map the local representation to the distributed one.

The key layer $\mathbf{k}$ has fixed activation and contains the generator for the inverse loci ($\mathbf{k}$ here corresponds to $\mathbf{k}^{-1}$ in Section 2, because $\mathbf{k}$ here is the decoding rather than the encoding key.).

The input representation is also local, only one input unit is active for each sequence. The outgoing weights from the single active input unit determine the activations at the code layer – these weights form a local-to-distributed map. The code layer injects activation into the hidden layer at the first step. After that the hidden layer activations are cycled through the context layer, so the input and code units no longer affect the network.

Each sequence is produced independently – no residual activations from the production of previous sequences remain in the network.

The HRN as described has no facility for detecting the end of the sequence. For the sake of simplicity, sequence lengths are supplied to the training and testing algorithms, and the nets are cycled the appropriate number of times. However, it is simple to incorporate end-of-sequence detection by including an extra output symbol that is defined to be the end-of-sequence indicator, and doing so does not result in any significant change in the capabilities of the net.

4.2 Unit functions

The HRN computes the following functions. The activations of the input, code and key layers remain constant during the production of a sequence. The parameter g is an adaptable input gain shared by all output units.

Code units:		$c_j = \sum_i a_i w^c_{ji}$	
Hidden units:	(first timestep)	$h_{1,j} = c_j$	
	(subsequent steps)	$h_{t,j} = \sum_i p_i k_{j-i}$	($\mathbf{h} = \mathbf{p} \circledast \mathbf{k}$)
Context units:		$p_{t+1,j} = h_{t,j}$	
Output units:	(total input)	$x_{t,j} = g \sum_i h_{t,i} w^o_{ji}$	
	(output)	$o_{t,j} = \frac{e^{x_{t,j}}}{\sum_i e^{x_{t,i}}}$	(softmax)

On all sequences, the net is cycled for as many timesteps as required to produce the target sequence. The outputs do not indicate when the net has reached the end of the sequence. However, other experiments have shown that it is a simple matter to add an extra output unit to indicate the end of the sequence.

The inputs and outputs to the network look like this:

		Output			
Sequence	Input	t = 0	t = 1	t = 2	t = 3
abc	1 0 0 00	1 0 0	0 1 0	0 0 1	
baac	0 1 0 00	0 1 0	1 0 0	1 0 0	0 0 1

4.3 Objective function

The objective function of the network is the asymmetric divergence between the activations of the output units ($o^s_{t,j}$) and the targets ($t^s_{t,j}$) summed over cases s and timesteps t, plus two weight penalty terms:

$$E = -\left(\sum_{stj} t^s_{t,j} \log \frac{o^s_{t,j}}{t^s_{t,j}}\right) + \frac{0.0001}{n}\sum_{jk}(w^c_{jk})^2 + \sum_j\left(1 - \sum_k (w^o_{jk})^2\right)^2$$

The first weight penalty term is a standard weight cost designed to penalize large weights (n is the number of hidden units). It is probably superfluous in the HRN, because the various operations from the w^c_{jk} to the total inputs of the output layer (x_j) are all linear, and scaling of the w^c_{jk} can be compensated for by the gain g. Later experiments without this weight cost term indicated that it has no effect on performance.

The second weight penalty term is designed to force the Euclidean length of the weight vector on each output unit to be one. This makes the output layer more like an ideal clean-up memory. The generalization performance with this special penalty term for the output weights is considerably better than that with a standard weight cost for the output weights (which is in turn better than that with no weight penalty). The reason for the worse performance with the standard weight cost is that the symbols which occur more frequently in the training set develop output weight vectors with higher magnitudes.

4.4 Derivatives

To do gradient descent learning, partial derivatives for all the adjustable parameters are required: $\frac{\partial E}{\partial w^o_{jk}}$, $\frac{\partial E}{\partial w^c_{jk}}$, and $\frac{\partial E}{\partial g}$. All of these can be easily calculated from the above equations using the chain rule and Rumelhart, Hinton, and William's [1986] unfolding-in-time technique. The only partial derivatives not encountered in common backpropagation networks are those for the activations of the context units:

$$\frac{\partial E}{\partial p_j} = \sum_k \frac{\partial E}{\partial h_j} y_{k-j} \qquad (\equiv \nabla_\mathbf{p} = \nabla_\mathbf{h} \circledast \mathbf{y}^*)$$

When there are a large number of hidden units it is more efficient to compute this derivative via FFTs as the convolution expression on the right ($\nabla_\mathbf{p}$ and $\nabla_\mathbf{h}$ are the vectors of partial derivatives of E with respect to $\mathbf{p}$ and $\mathbf{h}$).

4.5 Training networks and testing productive capacity

To be useful, a trainable storage scheme for structures must be able to store a wide variety of novel structures without requiring adjustment of its basic parameters after initial training. In an HRN, the basic parameters are the weights on the output units. The novel structures are sequences the network was not exposed to in the initial training. I refer to the ability of the network to store novel sequences as its *productive capacity*.

I performed initial training on a small number of short sequences in order to find good values for the basic parameters: the output weights w^o_{jk} and the output gain g. The initial training also finds code unit instantiations which produce the training sequences, which are determined by the weights w^c_{ij} coming from the block of input units on the left in Figure 3. To estimate the productive capacity of HRNs I freeze these parameters and test whether the network is able to produce longer novel sequences. For each novel sequence, I perform an independent search for a vector of code unit activations which will cause the network to produce the sequence. These activations are determined by the weights w^c_{ij} coming from the block of input units on the right in Figure 3.

For the initial training I use randomly chosen sequences of length 4 over various numbers of symbols. The number of symbols corresponds to the number of output units in the network. The number of sequences in the training set is 4 times the number of symbols, which means that on average each symbol occurs 16 times in a training set. For example, there are 12 sequences in the training set over three symbols. I initialize all weights to small random values, and the output gain to 1. I use conjugate gradient minimization and stop when all sequences were correctly produced (or when 400 steps have been taken). I judge a sequence to be correctly produced only if all its symbols are correct. I judge a symbol to be correct when the activation of the correct output unit exceeds 0.5 and exceeds twice any other output unit activation.

For estimating the productive capacity I use novel sequences of lengths 3 to 16 (32 sequences of each length). I use a separate conjugate gradient search to find a code for each sequence (starting from small random values), and stop when the sequence has been correctly produced, or when 100 steps have been taken.

5 Simple recurrent networks

I compare HRNs to other more conventional recurrent networks, both to get some idea of their relative capacities and because the comparison gives some insight into various reported difficulties with recurrent networks. Elman [1991] and Servan-Schreiber, Cleeremans and McClelland [1991] use Simple Recurrent Networks (SRNs) to process sequential input and induce small finite-state grammars. However, their training times are extremely long, even for very simple grammars. Maskara and Noetzel [1992] report similar results and make the plausible suggestion that the long training times are due to the difficulty of finding a recurrent operation that preserves information in the hidden units over many timesteps. Simard and Le Cun [1992b] report being unable to train a type of recurrent network to produce more than one trajectory through continuous space, a task which requires careful control over information in the hidden units.

Given these reported difficulties, I expected that SRNs would perform poorly at the sequence production task, since it requires preserving and transforming information in the hidden units over many timesteps. To test this, I built a version of the SRN for the sequence production task. The architecture of this SRN differs from the HRN in that the hidden layer receives activation through a weight matrix from the context layer, as with the SRNs. The network, which I refer to as a SRN, is shown in Figure 4.

The only difference between the SRN and the HRN is in the computation of the activations of the hidden units:

$$h_j = \tanh(c_j + \textstyle\sum_k w^r_{jk} p_k + b_j),$$

where b_j is a bias on each hidden unit. As with the HRN, the activations on the code units do not change during the production of a sequence. However, in contrast with the HRN, the activations of the code units flow to the hidden units at every timestep.

The objective function for the SRN is the same as for the HRN, except for the inclusion of an extra penalty term for large recurrent weights (i.e., w^r_{ij}, the weights from the context

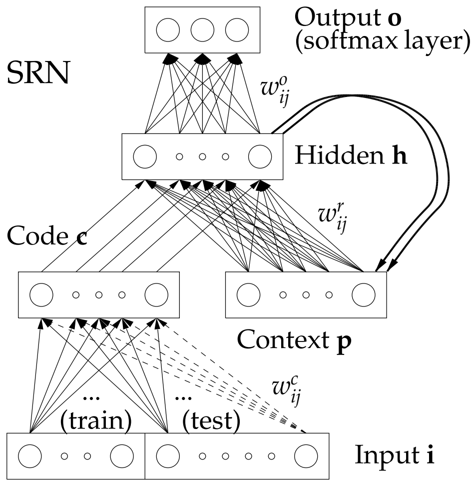

Figure 4: A simple recurrent network (SRN). The backwards curved arrows denote a copy of activations to the next timestep. The hidden, code, and context layers all have the same number of units. The left block of input units is used during training, the right block during testing.

to the hidden layer):

$$E = -\left(\sum_{stj} t^s_{t,j} \log \frac{o^s_{t,j}}{t^s_{t,j}}\right) + \frac{0.0001}{n}\left(\sum_{jk}(w^r_{jk})^2 + \sum_{jk}(w^c_{jk})^2\right) + \sum_j \left(1 - \sum_k (w^o_{jk})^2\right)^2$$

The SRNs described here differ in two important ways from other SRNs described in the literature (besides being used to produce rather than process sequences): the tanh function (symmetric sigmoid) is used on the hidden units, rather than the non-symmetric logistic function, and full unfolding-in-time is used to calculate correct derivatives, rather than Elman's truncated derivative scheme. Experiments showed that both changes significantly improved training times and productive capacity.

In the simulations, I use almost exactly the same training and testing regime as for HRNs. The only difference is that I experimented with larger training sets.

6 Training and productive capacity results

6.1 HRN results

HRNs prove to be easy to train, and to have good productive capacity. They also appear to have good scaling properties – as the number of hidden units increases they can store longer sequences or sequences over more symbols. Figure 5 shows the performance of HRNs with eight output units and varying numbers of hidden units. The points shown are averages of data from five runs, each starting with a randomization of all parameters. As the number of hidden units increases from 4 to 64 the productive capacity increases steadily. For a HRN with a fixed number of hidden units, the length of sequences that can be stored reliably drops as the number of symbols increases, as shown in Figure 6. This is expected because when there are more symbols, more information is required to specify

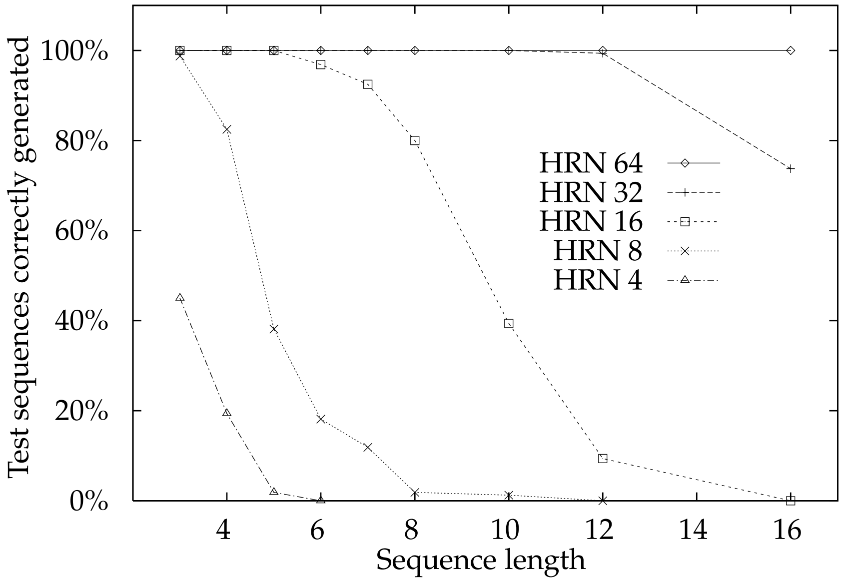

Figure 5: Percentage of novel sequences that can be produced, versus length of those sequences for HRNs with varying numbers of hidden unit. The sequences are over 8 symbols. All points are averages of data from 5 different training runs.

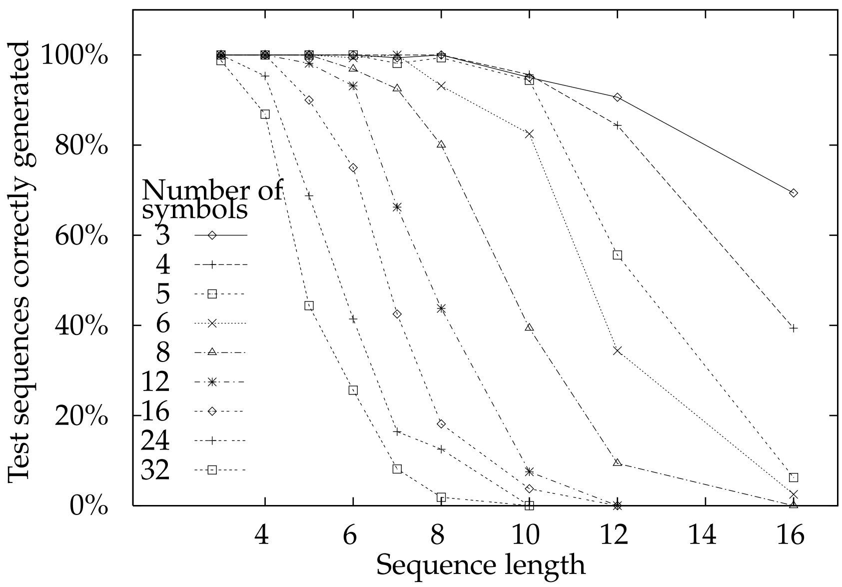

Figure 6: Percentage of novel sequences that can be produced by HRNs with 16 hidden units, versus length of those sequences, for varying numbers of symbols (i.e., varying numbers of output units). All points are averages of data from 5 different training runs.

them. For example, the same amount of information is required to specify a sequence of length eight over three symbols as a sequence of length four over nine symbols.

To make a rough guess at the information efficiency of HRN sequence codes we can calculation how much information is in the maximum-length sequence that can be reliably encoded. For example, Figure 5 shows that a HRN with 16 hidden units can reliably store sequences of length 5 over 8 symbols. Such a sequence contains 15 bits of information (assuming all symbols are equally likely), which gives an information efficiency of nearly 1 bit per code element (15/16). This figure appears stable for different numbers of hidden units and for different numbers of symbols. This compares well with the 0.16 bit per element achieved by random vector circular convolution (Section 2.4). The increase in capacity can be attributed largely to the adaptive method of forming codes.

6.2 SRN results

SRNs turned out to be better at this task than expected. Figure 7 shows how the performance varies with sequence length for various networks with 16 hidden units. SRNs required much more training data, which is not surprising since they have more parameters in the form of weights from the context to the hidden layer. The productive capacity for a SRN trained on 12 sequences of length 4 (the same amount of data as a HRN) was very poor – compare the bottom and top lines in the graph. However, with more training data consisting of longer sequences, the productive capacity was better, though never as high as that of HRNs. In the Figure, "SRN+" is the productive capacity of an SRN trained on 48 sequences of length 8 (8 times as much training data) and "SRN++" is one trained on 192 sequences of length 8 (32 times as much training data). The length of training sequences appeared to have a significant effect on productive capacity. Experiments not shown here indicated that the increase in productive capacity obtained by doubling the length of training sequences (4 to 8) was greater that obtained by doubling the number of training sequences.

Interestingly, freezing the recurrent weights at random values results in a significant increase in productive capacity for a SRN trained on the small training set ("SRNZ" in the Figure). Random values for all the weights coming into the hidden units are chosen from a Gaussian distribution with mean zero and the same variance as those from a sample trained SRN (0.2 for a net with 16 hidden units). These weights are frozen at the beginning of training and never changed. The productive capacity achieved with these weights tends to suggest that the recurrent weights are not doing much more than a random mapping. The performance would probably improve further if the weight matrix were chosen so that all its eigenvalues were of equal magnitude (like the recurrent operation in HRNs).

In general, modifications to SRNs which make them more like HRNs seem to improve performance. As mentioned before, early experiments with the standard non-symmetric sigmoid operation on the hidden layer had very poor performance. The symmetric sigmoid (tanh), which is more like the linear transfer function in the HRN, gives much better performance. Freezing the hidden weights at random values also improves performance (for nets with a small training set). Other experiments showed that removing the bias on the hidden units usually improves performance. Changing the hidden unit transfer function of the SRN from tanh to linear improves performance on shorter novel sequences, but not on longer novel sequences. This is probably because the linear transfer function allows eigenvectors with large eigenvalues to grow exponentially, whereas the tanh function limits their growth. The only change that makes the SRN more like the HRN, and that consistently

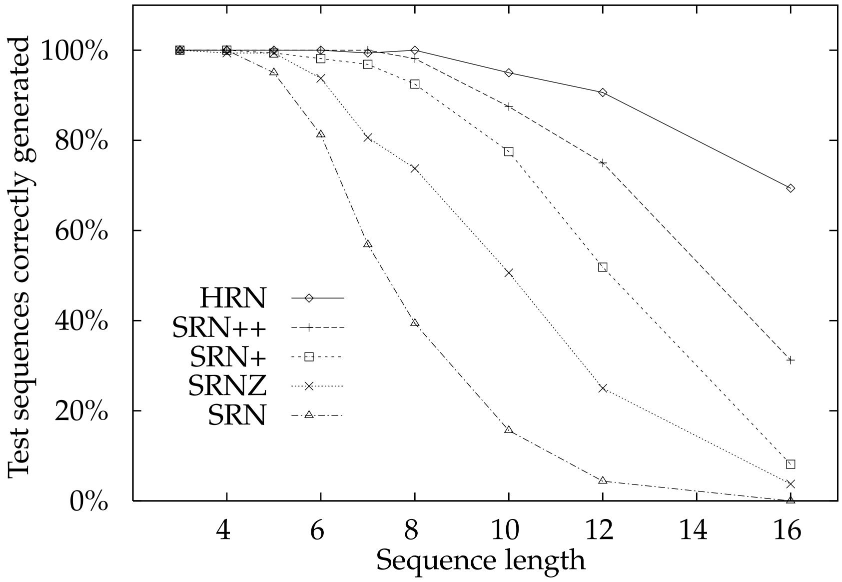

Figure 7: Percentage of novel sequences that can be produced, versus length of those sequences, for various types of networks (all with 16 hidden units, and 3 symbols). "SRNZ" is the SRN with frozen random recurrent weights. "SRN+" is the SRN with 8 times as much training data and "SRN++" is the SRN with 32 times as much. All points are averages of data from 5 different training runs.

harms performance, is making the code units inject activation into the hidden units at the first timestep only.

6.3 General training and testing issues

The training times for both the HRNs and the SRNs are very short. Both require around 30 passes through the training data to train the output and recurrent weights. Finding a code for test sequences of length 8 takes the HRN an average of 14 passes, and the SRN an average of 57 passes (44 with frozen recurrent weights). The SRN trained on more data takes much longer for the initial training (average 281 passes) but the code search is shorter (average 31 passes).

Local minima in the space of codes for trained networks are not a significant problem for either HRNs or SRNs. Experiments with restarts for failed searches for a sequence code (during testing) show that allowing one restart improves the productive capacity figures by only 0.3% for HRNs and 1.3% for SRNs.

7 Trajectories in continuous space

Simard and Le Cun [1992b] devise a "Reverse Time-Delay Neural Network" for producing trajectories through a continuous space. They train the network to produce pen trajectories for writing the characters "A" to "Z". One reported motivation for the complex architecture of their network is the difficulty they had with training a simpler recurrent network to produce more than one trajectory. HRNs can be adapted to produce trajectories, and have a much simpler architecture than Reverse Time-Delay Neural Networks. Only two modifications need be made: (a) change the function on the output units to sigmoid and add biases, and (b) use a fractional power of a random vector for the key vector. The hidden unit activations are still $\mathbf{k} \circledast \mathbf{c}$, where $\mathbf{c}$ is the code vector. The fractional power key vector $\mathbf{k}$ is derived by taking a random unitary vector $\mathbf{x}$ and multiplying the phase angle of each frequency component by some fraction α, i.e., $\mathbf{k} = \mathbf{x}^{\alpha}$. The result is that $\mathbf{k}^i$ is similar to $\mathbf{k}^j$ when the difference between i and j is less than $1/\alpha$, and the similarity is greater when i and j are closer. This is good for generating smooth trajectories because the decoding key at one step is similar to the decoding key at the next step. This means that the contents of the hidden layer will be similar at successive timesteps. The rate at which the output trajectory is traversed can be adjusted by changing α – smaller values result in a slower rate of change.

I tested a trajectory producing HRN on a task similar to Simard and Le Cun's: producing pen trajectories for the digits zero to nine. The HRN has 16 hidden units and a key vector with $\alpha = 0.06$. The trajectories have 100 steps in three dimensions (X, Y, and pen up/down). As with other HRNs, there are two sets of parameters: the input-to-code weights, and the output weights. I trained the network on 20 instances of handwritten digits and then froze the output weights. With the output weights frozen, the network could easily find codes that produce trajectories for novel digit instances. Figure 8 shows the X and Y target trajectories and the output of the network for one instance. The HRN does not reproduce trajectories exactly – the code has a very limited number of parameters and can be expected to impose some sort of regularization on the trajectory.

The code that produces a trajectory is a 16-element vector. One would expect the codes which produce similar trajectories to be similar. This does seem to be the case: 40 codes for producing the trajectories for 40 digits clustered into the digit classes with only one exception. The code vector is potentially useful wherever a compact representation for trajectories is needed, e.g., in classification.

It is possible to vary the rate of trajectory traversal by adding a unit to the network which exponentiates the key vector. Let ζ be the output of this unit. The hidden layer activations will be $\mathbf{k}^{\zeta} \circledast \mathbf{c}$. The network can be set up so that the value of ζ changes during the production of a trajectory. The base-line value for ζ should be one. To progress more rapidly through the trajectory ζ should be greater than one, and to progress more slowly, ζ should be less than one. It is not difficult to calculate partial derivatives for ζ, and preliminary experiments showed that a HRN could learn to control ζ dynamically so as to align the network output with a target sequence.

8 Hierarchical HRNs

A significant attraction of circular convolution as an associative memory operator is its potential for representing hierarchical structure. It is simple to construct hierarchical representations for sequences using convolution-based associative memory [Plate 1994]. In order to investigate whether gradient descent training could be used to find hierarchical codes for sequences I built a two-level HRN. It consists of two HRNs with the outputs of one (the main HRN) supplying the code for the other (the sub-HRN). Figure 9 shows the overall architecture of this system. The subsequence decoder (at the top of the figure) produces sequences of symbols from the codes it receives from the main sequence generator. When the subsequence decoder has finished a subsequence it signals the main sequence

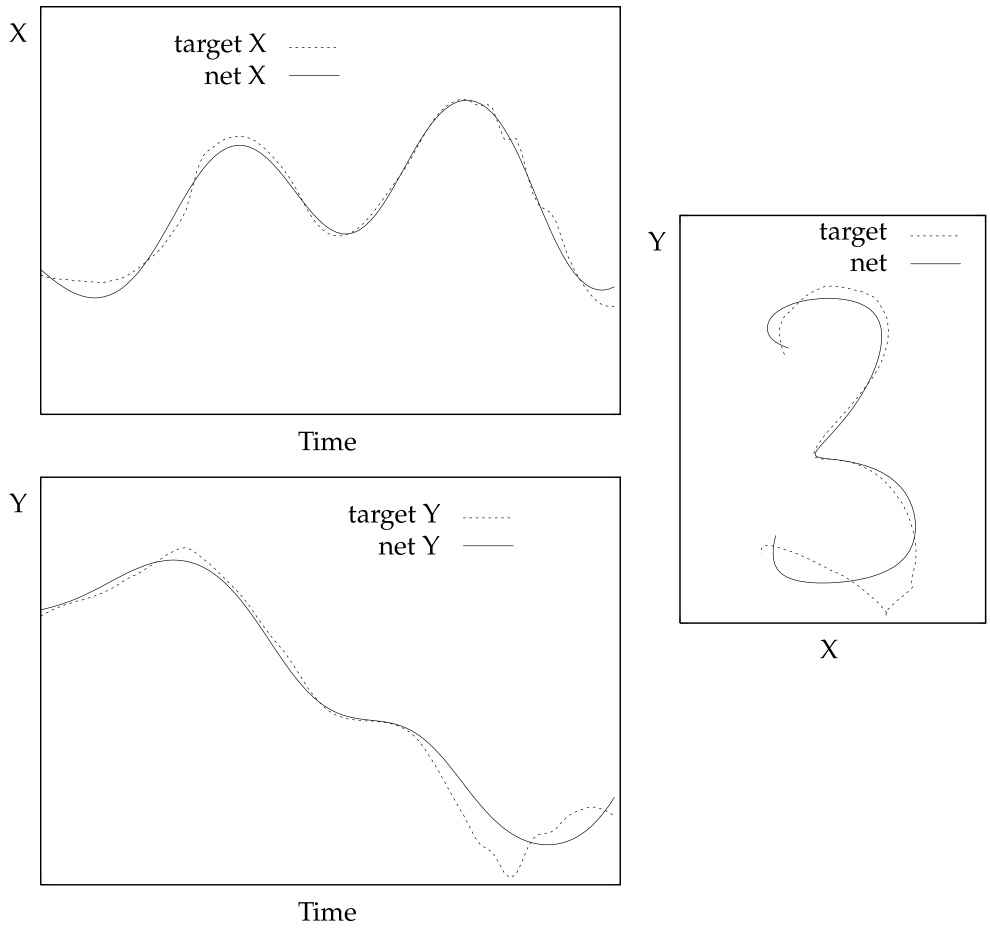

Figure 8: Targets and outputs of a HRN trained to produce trajectories through continuous space. The target trajectory was not one that the network was trained on.

decoder to produce the next subsequence code. Each decoder is similar to one of the HRNs already described. The clean-up memory for the subsequence decoder, i.e., its output units, contains symbol codes. The clean-up memory for main sequence decoder contains subsequence codes. The clean-up memory uses a local representation, so the output of the main sequence decoder must be transformed back to a distributed representation (the subsequence codes) by a local-to-distributed mapping.

For example, suppose the encoding key for subsequence generator is **k**, and the key for the main sequence generator is **j**. Suppose that "abc", "de", and "fgh" are chunks the main sequence generator knows about. Then the sequence "abcdefgh" can be stored as

$$(\mathbf{a} + \mathbf{b} \circledast \mathbf{k} + \mathbf{c} \circledast \mathbf{k}^2) + (\mathbf{d} + \mathbf{e} \circledast \mathbf{k}) \circledast \mathbf{j} + (\mathbf{f} + \mathbf{g} \circledast \mathbf{k} + \mathbf{h} \circledast \mathbf{k}^2) \circledast \mathbf{j}^2.$$

This two-level HRN was able to produce sequences given appropriate codes. However the network was not able to find main sequence codes by using gradient descent with backpropagated errors from target symbols – there were too many local minima. This was the case even when the components of the network were trained separately and frozen. The local minima are most likely due to the presence of an intermediate clean-up memory.

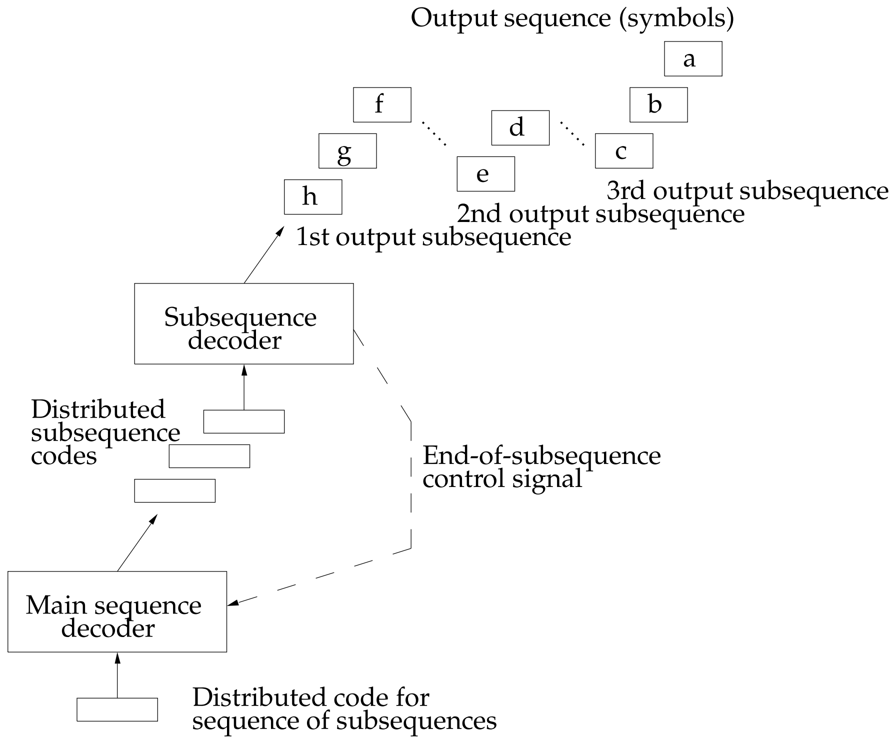

Figure 9: A two-level system for decoding nested sequences. The main system outputs a series of codes for subsequences, which are input to the subsequence decoder. When the subsequence decoder has completed a subsequence it signals the main sequence decoder to produce the next subsequence.

9 Discussion

9.1 Suggestions for improving the ability of recurrent networks to store context

The experiments described in this paper show that it is not difficult for recurrent networks to learn fixed-width codes for sequences. This, and the theory of trajectory association suggest two possible means for improving the ability to store context in recurrent networks for sequence recognition: (1) include the additional constraint that the net should reproduce recent context (as Maskara and Noetzel [1992] do), or (2) constrain the recurrence operation to have eigenvalues of magnitude near to one. This could be done in several ways: using convolution with a unitary vector as the recurrence operation, or by a soft constraint on the magnitudes of eigenvalues. It should be noted that this is not an entire solution to the problem of storing *relevant* context for recognition, which will usually only be a summary of the total context.

9.2 Application to processing sequential input

An issue in processing sequential data with neural networks is how to present the inputs to the network. One approach has been to use a fixed window on the sequence, e.g., as in Sejnowski and Rosenberg's [1986] NETtalk. A disadvantage of this is that any fixed size of window may not be large enough in some situations. Instead of a fixed window, Elman [1991] and Cleeremans *et al* [1991] used a recurrent net to retain information about previous inputs. A disadvantage of this is the difficulty that recurrent nets have in retaining information over many timesteps. HRNs offer another approach: use the codes that produce a sequence as input rather than the raw sequence. This would allow a fixed-size network to take sequences of variable length as inputs (as long as they were finite), without having to use multiple input blocks or windows. This would not be immediately applicable to networks which process sequences with no clearly defined beginning or end, but some modification could be made along the lines of using a code which produces the recent relevant inputs.

9.3 Application to classification of sequences

The codes that produce sequences have similarity structure, e.g., the code for producing "abc" is more similar to the code for "acc" than the code for "cab". This also applies to the codes for producing trajectories through continuous space.

9.4 Improving the capacity of convolution-based memory

Adapting the representations of objects and traces gave a five-fold increase in the storage capacity of the trajectory-association method. Experiments with networks having many more output units showed that the information capacity was not affected by the number of different symbols used in the sequences.

The improvement in capacity is encouraging because it suggests that the capacity of other types of convolution-based memory could be improved by adapting representations, and also that traces that fail to decode correctly could be corrected by a simple gradient descent procedure.

10 Conclusions

It is straightforward to build the decoding operations for the trajectory-association storage method into a recurrent network. The resulting HRN can be trained using standard neural network methods. During training, the network simultaneously discovers memory traces for sequences and good representations for the objects. Experiments with HRNs demonstrate four points:

- Gradient descent techniques are fast and reliable for learning representations for non-nested sequences from random starting points.
- Gradient descent techniques are impractical for learning representations for nested sequences from random starting points.
- Adapting memory traces and object representations increases the storage capacity of convolution-based memories.
- For the task of producing sequences and trajectories, HRNs are as good as, if not better than, more conventional recurrent networks. They have fewer parameters, and consequently require less training data.

The convolution operation appears to be ideal for the recurrent operation in situations where information must be preserved but transformed in the hidden layer, so as not to be confused with newer information, because all the eigenvalues of the recurrent operation have a magnitude of one.

References

Davis, P. J. 1979. *Circulant matrices*. John Wiley & Sons, New York.

Elman, J. 1991. Distributed representations, simple recurrent networks and grammatical structure. *Machine Learning*, 7(2/3):195–226.

Hinton, G. E. 1986. Learning distributed representations of concepts. In *Proc. Eighth Annual Conference of the Cognitive Science Society*, pages 1–12. Hillsdale, NJ: Erlbaum.

Hopfield, J. J. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences U.S.A.*, 79:2554–2558.

Maskara, A. and Noetzel, A. 1992. Forcing simple recurrent neural networks to encode context. In *Proceedings of the 1992 Long Island Conference on Artificial Intelligence and Computer Graphics*.

Plate, T. A. 1994. *Distributed Representations and Nested Compositional Structure*. PhD thesis, Department of Computer Science, University of Toronto.

Plate, T. A. 1995. Holographic reduced representations. *IEEE Transactions on Neural Networks*. To appear.

Rumelhart, D. E., Hinton, G. E., and J., W. R. 1986. Learning internal representations by error propagation. In *Parallel distributed processing: Explorations in the microstructure of cognition*, volume 1, pages 318–362. The MIT Press, Cambridge, MA.

Sejnowski, T. J. and Rosenberg, C. R. 1986. *NETtalk: A parallel network that learns to read aloud*. Technical report 86-01, Department of Electrical Engineering and Computer Science, Johns Hopkins University, Baltimore, MD.

Servan-Schreiber, D., Cleeremans, A., and McClelland, J. L. 1991. Graded state machines: The representation of temporal contingencies in simple recurrent networks. *Machine Learning*, 7(2/3):161–194.

Simard, P. and LeCun, Y. 1992a. Reverse tdnn: an architecture for trajectory generation. In Moody, J. M., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4 (NIPS*91)*, Denver, CO. Morgan Kaufman.

Simard, P. and LeCun, Y. 1992b. Reverse TDNN: an architecture for trajectory generation. In Moody, J. M., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4 (NIPS*91)*, Denver, CO. Morgan Kaufmann.