Using Simple Recurrent Networks to Learn Fixed-Length Representations of Variable-Length Strings

Christopher T. Kello
Daragh E. Sibley
Andrew Colombi

Department of Psychology
George Mason University
Fairfax, VA 22030
ckello@gmu.edu, dsibley@gmu.edu

Abstract

Four connectionist models are reported that learn static representations of variable-length strings using a novel autosequencer architecture. These representations were learned as plans for a simple recurrent network to regenerate a given input sequence. Results showed that the autosequencer can be used to address the dispersion problem because the positions and identities of letters in a string were integrated over learning into the plan representations. Results also revealed a moderate degree of componentiality in the plan representations.

Linguistic structures vary in length. Paragraphs contain varying numbers of sentences, sentences contain varying numbers of words, and words contain varying numbers of letters and sounds. By contrast, standard connectionist mechanisms are designed to compute fixed-length vectors. In connectionist models of human language processing, variable-length structures have been most commonly processed using simple recurrent networks (Elman, 1990; Jordan, 1986)

Perhaps the most well-known usage of SRNs is to learn the transitional probabilities in a grammar (Christiansen et al., 1998; Cleeremans et al., 1989). The SRN is given a sequence of inputs, and it is trained to predict the next input on the basis of the current and previous inputs. Given this task, the fixed-length hidden representations learn to hold just enough information about the variable-length sequence of past inputs in order to best predict the next input. However, the hidden representations are not pressured to hold enough information such that they could be used to *reproduce* the past N inputs in order.

Two other common usages of SRNs are 1) to map static, fixed-length inputs onto variable-length outputs, or 2) to map variable-length inputs onto static, fixed-length outputs. As an example of the former, Dell and his colleagues (Dell et al., 1993) built an SRN model of phonological encoding in which a fixed-length representation of a word lemma or word form served as a "plan" representation to generate a variable-length sequence phonological features. As an example of the latter, Plaut and Kello (1999) built a model of phonological development in which variable-length sequences of phonetic features were mapped onto fixed-length semantic representations. In both of these examples (and all such previous work), the static representations were engineered by the modelers.

The Autosequencer

Here we present an extension of the SRN architecture, termed an *autosequencer*, that is able to learn fixed-length representations of variable-length strings (the architecture originated as tool to interpret the outputs of the model presented by Plaut and Kello. 1999). The autosequencer is motivated by the need to learn representations that simultaneously encode information about the identities and positions of characters in variable-length strings.

In the context of lexical processing, Plaut et al. (1996) described the tension between identity and position as the *dispersion* problem. Both the identities and positions of letters and sounds must be encoded in order to comprehend and produce written and spoken words, respectively. Connectionist models of lexical processing typically devote units to represent letters or sounds in particular positions, such as a unit to code P or /p/ in the onset of a word. The consequence of this slot-based coding scheme is that knowledge about P or /p/ in the onset is represented separate from knowledge about P or /p/ in the end of a word. Moreover, slot-based codes work only for strings whose lengths are relatively uniform, i.e., they have been used for monosyllabic, but not multisyllabic, words. Wickelfeatures are an alternative to slot-based codes (e.g., Seidenberg and McClelland, 1989), but they too are limited in important ways (Pinker and Prince, 1988).

The autosequencer architecture presented here addresses the dispersion problem. Analogous to an autoencoder, representations are learned to reproduce inputs as outputs. The difference is that these learned representations serve as plans for reproducing variable-length input strings. We show that this pressures the autosequencer to learn representations that code both position-dependent and position-independent information about letters in variable-length strings. These representations exhibited a limited degree of compositionality.

Training and Testing Corpus

The alphabet consisted of five letters. These letters were artificial, but let us refer to them as A through E for the sake of discussion. The space of inputs and outputs was defined by all possible letter sequences from one to five letters in length (3905 possible sequences). Each sequence was terminated by a stop character. Each letter was represented as a 3-bit binary pattern. Four different subsets of strings were selected as training sets for four different autosequencers. The *two-fifths* training set was created by choosing at random two out of every five strings at each of the five lengths (1562 total). The *no-fours* training set was created by removing all four-letter strings from the two-fifths training set (1312 total). The *four-five* training set was created by selecting only the four-letter and five-letter strings from the two-fifths training set (1500 total). The *two-ends* training set was created by choosing all sequences whose ending letter was either an A or B (not including the stop character; the two end letters were chosen arbitrarily).

Model Architecture

The architecture of the autosequencer is diagrammed in Figure 1. Letter patterns were presented sequentially on the input units, and the network was trained to reproduce the pattern sequence over the output units. Over a given input sequence, a representation was built up on the static layer, up to and including the final stop character. The decoder was not run until the stop character. At that point, the final static representation was held constant, and the decoder side of the network was run to produce a sequence of outputs. The numbers of hidden units were chosen on the basis of trial-and-error to be close to the minimums necessary to learn the training sets to criterion.

The trick to this architecture is in how to back-propagate error from the encoder side to the decoder side. Cross-entropy error signals (Rumelhart et al., 1995) were generated on each forward propagation of the decoder, and the signals were back-propagated to the static layer only. Error derivates were accumulated at the static layer over the forward pass of the decoder, up to and including the last forward propagation for the stop character.

At that point, the forward pass of the encoder was “replayed”, and the accumulated derivatives were used as error signals. Specifically, these derivatives were back-propagated in time from the last to the first forward propagation of the encoder.

One informative point about this method of training is that accumulated derivatives on the static layer act as “moving targets” for the encoder, because they change as weights in the decoder change. Also, activations on the static layer act as “moving inputs” to the decoder, because they change as weights in the encoder change. Our experience has been that this interdependence can destabilize learning in larger, more difficult training sets. We have found that learning can be made more stable by making the encoder learning rate slower than the decoder learning rate. This was not needed for the current simulations, however.

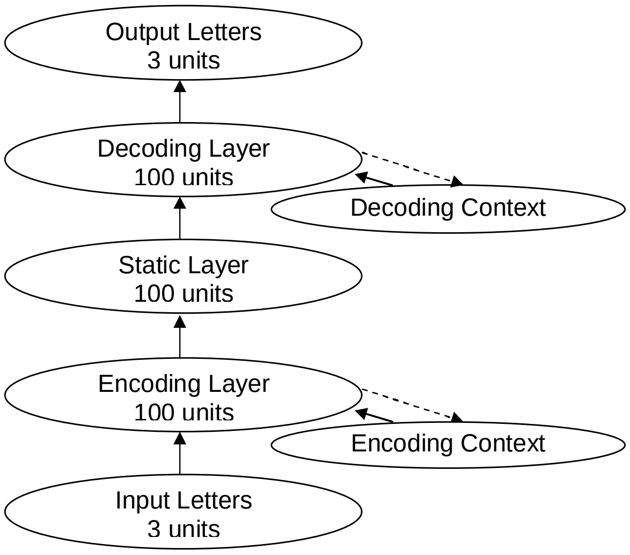

Figure 1. Architecture of the Autosequencer

Input unit activations were set to 0 or 1, and all other unit activations were computed as the hyperbolic tangent (sigmoid) of their net inputs. Net inputs were calculated as the dot product of the incoming weight vector and the activation vector over the sending units. Context units were copies of the corresponding hidden units from the previous time step (initialized to zero).

Training Procedure

For each of the four autosequencers, items were chosen at random from the training set, and weight updates were made after weight derivatives were accumulated over 200 training items. Weight derivatives were calculated with the learning rate fixed at 0.0001, and a momentum parameter fixed at 0.8. After 2700 epochs of weight updates, the activations of all output units for all letters of all training sequences were on the correct side of zero. Model performance did not change substantially with further training, so results are presented up to 2700 epochs. To be consistent, the other three models were also trained for a total of 2700 epochs.

Simulation Results

In Figure 2, the percentage of items correct is plotted as a function of training for the two-fifths model. Percentages are plotted separately for trained items and novel items of lengths 1-3 versus 4-5. Novel items included all untrained sequences.

Accuracies for trained items show that the autosequencer was successful at learning static representations for variable-length strings. Accuracies for novel items show that learning generalized to both short and long sequences, but generalization was better for longer sequences. This difference occurred because, unavoidably, there were many more longer sequences to train on, compared with shorter ones. Hidden unit sizes of up to 500 were tested (results not shown here), and generalization improved somewhat with more units. However, performance on the longer novel sequences was always better than performance on the shorter ones.

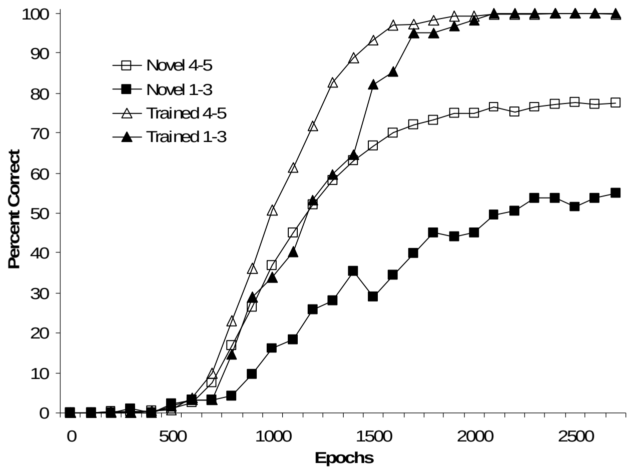

Figure 2. Accuracies for the Two-Fifths Model

A basic question about the autosequencer is how learning and representations differed as a function of position in a string. Two analyses were conducted to address this question. In one, errors were tallied as a function of position for all strings of lengths 2-5, across all epochs of training. In the other, similarities of the learned static representations were compared at the end of training for strings that differed only by one letter in the Nth position, where N was all positions for strings of lengths 2-5. Similarity was measured by the normalized dot product (NDP).

The results of these two analyses are shown in Table 1. One can see that, in general, the static layer coded the beginning positions more distinctly than the end positions: there were fewer errors in processing letters in the first position, and static representations for strings that differed in the first position were less similar to each other, compared with differences in other positions. For longer strings, the end positions were coded more distinctly compared with the middle positions. It is important to note that these differences were caused by the autosequencer itself; there were no biasing statistics in the training corpus that could have caused them.

Table 1. Percentages of errors and mean NDPs as a function of length and position for the two-fifths model.

	% of Errors String Length				Similarity (NDP) String Length			
	2	3	4	5	2	3	4	5
Pos								
1	32	22	16	19	.05	.13	.15	.42
2	88	61	40	29	.76	.73	.79	.75
3		53	49	46		.81	.83	.84
4			43	40			.87	.80
5				31				.77

A more pointed question is, how did the static representations code the positions and identities of letters? A number of tests were conducted to address this question. First, we counted the number of errors during training in which a letter was inserted, deleted, or transposed. These types of errors demonstrate some degree of separation in the coding the positions and identities of letters, a property that is lacking in more traditional connectionist representations (e.g., slot-based codes).

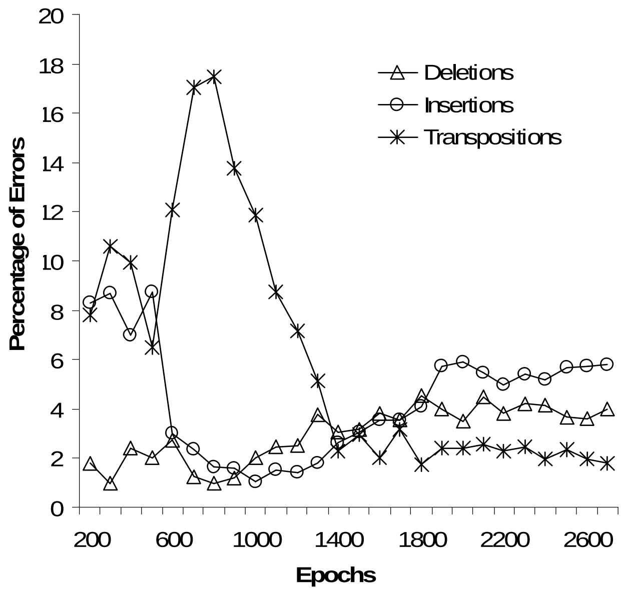

Figure 3. Three Error Types for the Two-Fifths Model

In Figure 3, error types are plotted as a function of training for all 3092 possible strings up to 5 characters long. The graph shows that the autosequencer made all three types of errors, which demonstrates that position and identity information were coded at least somewhat separately from each other. Interestingly, there was a spike in transposition errors midway through training. It appears that the network went through a period in which position information was particularly weak, but further analyses are needed to test the hypothesis.

Another way to test for the separation of position and identity is to examine the similarity structure of the learned static representations. We conducted two analyses of the similarity structure, one to test for coding of identity separate from position, and the other to test for coding of position separate from identity.

To test coding of letter identity, NDPs were calculated for strings that share letters in different positions. For instance, the letter strings AYE and YEA share three non-overlapping letters in common. This test was performed on all pairs of three-letter strings that shared 0 to 3 non-overlapping letters in common. None of these pairs shared letters in the same positions. If the static representations coded letter identities at least partially independent of their position, then similarity should increase as the number of non-overlapping letters in common increases. Results confirmed this prediction: the mean NDPs were –0.14, –0.11, –0.08, and –0.05 for 0 to 3 letters in common, respectively. While the overall low similarities suggest that position information was very important, the linear increase in similarity stands as clear evidence that the letters were coded similarly across positions in the static representations.

To test coding of letter position, similarities of same-length strings were measured against similarities of comparable different-length strings. If the static representations coded position information at least partially independent of letter identities, then same-length strings should more similar to each other. We tested this prediction by first calculating NDPs for all pairs of three-letter strings that differed in only one position. Three such pairs might be CAT-BAT, ACT-ANT, and BID-BIN. We then calculated NDPs for three-letter strings paired with four-letter strings that contained the paired three-letter string. Three such pairs might be HAT-CHAT, WIN-WINE, and ACT-FACT. The mean NDP for same-length strings was 0.55, whereas it was only 0.23 for different-length strings. This difference occurred despite the fact that different-length strings shared more letters in common (3 versus 2 in the same-length pairings). This result confirms that the static representations coded the positions of letters at least partially independent of their identities.

While positions and identities may be coded separately to some degree, it is unclear how well the autosequencer can generalize on the basis of these codes. The analyses graphed in Figure 2 show that generalization is fairly good for string lengths and letter positions that appeared in the training corpus. However, to show full compositionality in the learned static representations, the autosequencer would need to generalize to string lengths and letter positions outside the training corpus.

Three autosequencers were trained to test compositionality of the static representations. The no-fours model served to test whether learning could be interpolated to four-letter strings when only shorter and longer strings are trained. The four-five model served to test whether learning could be extrapolated to sequences shorter or longer than those trained. Results with these two models showed no tendency to interpolate or extrapolate. Specifically, none of the four-letter strings were processed correctly by the no-fours model, and none of the three-letter or six-letter strings were processed correctly by the four-five model. These results show some of the limits of compositionality in the current autosequencers, but more simulations are necessary to confirm these limits.

The third and final model tested whether individual letters could be correctly sequenced in untrained positions. In the two-ends model, three of the five letters never appeared at the end of a sequence in the training set. After training, 15% of the novel sequences with untrained end-letters were processed correctly. This modest generalization suggests that the autosequencer has the potential to sequence letters in untrained positions, but more simulations are necessary to determine whether this potential can be fully realized.

Conclusions

A variety of hybrid mechanisms have been proposed to enhance the componentiality of connectionist processing (Browne and Sun, 1999; Plate, 1994; Pollack, 1990; Shastri and Ajjanagadde, 1993). The autosequencer presented here uses SRNs to learn distributed representations that code the position and identity of elements in a variable-length sequence.

The autosequencer is similar to a sequential recursive auto-associative memory (RAAM Pollack, 1990). A RAAM can learn to code a variable-length string as a left-branching binary tree. To do so, it must learn a representation that can decode each branch of the tree. For instance, to learn a code for the letter-string TRAP, it would also learn codes for the strings T, TR, and TRA. By contrast, the autosequencer is not forced to learn sequences as left-branching binary trees, and it is not forced to learn representations that can decode parts of strings in its training corpus. Further work is necessary to compare the RAAM and autosequencer architectures in order to determine the consequences that their similarities and differences may have on processing capabilities.

The autosequencer simulations reported here demonstrated a limited degree of compositionality. On the one hand, representations were learned to integrated knowledge about the identities and positions of letters in variable-length strings. This knowledge generalized fairly well to the sequencing of novel strings, so long as their lengths and letter positions were represented in the training corpus. On the other hand, learning did not generalize equally to all sequence lengths, and there was no evidence of interpolation or extrapolation to string lengths outside the training corpus.

More work is necessary to determine whether these shortcomings are inherent to the autosequencer, or whether they were partly due to choices in the model parameters and representations. But putting these possible shortcomings aside, the autosequencer was designed to learn representations of variable-length linguistic structures. Our plan is to "port" such representations into connectionist models of language processing, such as models of word reading (e.g., see Kello and Plaut, 2003). The current simulations demonstrate the viability of this plan.

Acknowledgements

This work was funded in part by NIH Grant MH55628, and NSF Grant 0239595. The computational simulations were run using the Lens network simulator (version 2.6), written by Doug Rohde (http://tedlab.mit.edu/~dr/Lens). We thank David Plaut for collaborations on precursors to this work.

References

Browne, A. and Sun, R., 1999. Connectionist variable binding. *Expert Systems*, 16: 189-207.

Christiansen, M.H., Allen, J. and Seidenberg, M.S., 1998. Learning to segment speech using multiple cues: A connectionist model. *Language & Cognitive Processes*, 13(2&3): 221-268.

Cleeremans, A., Servan-Schreiber, D. and McClelland, J.L., 1989. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3): 372-381.

Dell, G.S., Juliano, C. and Govindjee, A., 1993. Structure and content in language production: A theory of frame constraints in phonological speech errors. *Cognitive Science*, 17(2): 149-195.

Elman, J.L., 1990. Finding structure in time. *Cognitive Science*, 14(2): 179-211.

Jordan, M.I., 1986. Serial order: A parallel distributed processing approach. 8604 ICS Technical Report, University of California at San Diego, La Jolla, CA.

Kello, C.T. and Plaut, D.C., 2003. Strategic control over rate of processing in word reading: A computational investigation. *Journal of Memory & Language*, 48(1): 207-232.

Pinker, S. and Prince, A., 1988. On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28(1-2): 73-193.

Plate, T.A., 1994. Distributed representation and nested compositional structure, Department of Computer Science, University of Toronto, Toronto, CA.

Plaut, D.C., 1996. Understanding human performance in quasi-regular domains: Insights from connectionist modeling of normal and impaired word reading. *International Journal of Psychology*, 31(3-4): 1003-1003.

Plaut, D.C. and Kello, C.T., 1999. The emergence of phonology from the interplay of speech comprehension and production: A distributed connectionist approach. In: B. MacWhinney (Editor), *The emergence of languag*e. Erlbaum, Mahweh, NJ, pp. 381-415.

Pollack, J.B., 1990. Recursive distributed representations. *Artificial Intelligence*, 46: 77-105.

Rumelhart, D.E., Durbin, R., Golden, R. and Chauvin, Y., 1995. Backpropagation: The basic theory, Chauvin, Yves (Ed); Rumelhart, David E (Ed) (1995) Backpropagation: Theory, architectures, and applications (pp 1-34).

Seidenberg, M.S. and McClelland, J.L., 1989. A Distributed, Developmental Model of Word Recognition and Naming. *Psychological Review*, 96(4): 523-568.

Shastri, L. and Ajjanagadde, V., 1993. From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings. *Behavioral & Brain Sciences*, 16: 417-494.