On Sequential Learning

Isao Inoue

1. Introduction

Many types of human cognition involve temporally ordered flow of events where the order of events plays a crucial role in executing tasks in an appropriate way. For example, the information on the sequential order of words employed in a sentence is essential for the correct interpretation of the sentence. When we process a word in a sentence, the availability of information about what words previously occur in the sentence is also crucially important. Thus, the important problem about computational models for simulating the acquisition of regularities in such sequential processing is developing internal context representations relevant to the processing of the current input. In this paper, we examine connectionist models that learn to encode temporal context information. The two models that we explore here fall under the category of a simple recurrent network or SRN, which is basically a three-layer network involving a special context layer that recursively provides contextual information that can be used to optimize processing of current input, as shown below.

![Simple Recurrent Network Diagram]

Figure 1. The simple recurrent network used in the simulations. Each rectangle represents a layer of units, and the number of units in each layer is indicated by the side. All the forward-going connections are trainable. The downward arrow from the hidden layer to the context layer denotes a simple copy operation and is not subject to training.
In the SRN shown in Figure 1, the hidden layer receives information about the current element from the input layer, and concurrently obtains its own previous processing result via the context layer with a delay of one time step. When this process is carried out recurrently, the SRN can integrate the current input with the contextual information accumulated through the processing of entire sequential elements previously presented to the network.

We compare performance of two SRN models, namely Elman’s sequence prediction network (Elman (1990)), abbreviated here as Elman network, and Leabra network (O’Reilly and Munakata (2000: chapter 6)) with respect to the capacity to use nonlocal contingencies between elements of sequences generated by a finite state automaton (FSA) or a finite state grammar. FSA has been used extensively in artificial grammar (AG) learning research (Reber (1993)). It has been shown that human subjects, after being exposed to sentences generated by FSA, acquire implicit knowledge of the regularities in the generating grammar and would be able to sort new sentences generated by the grammar from those that violate it (Cleeremans (1993), and Cleeremans and McClelland (1991)). In recent experimental studies, it is also demonstrated that infants could become sensitive to grammatical regularities inherent in FSA with minimum exposure (for example, two minutes) to exemplar sentences and display such ability by discriminating grammatical strings from ungrammatical ones (Gomez and Gerken (1999)).

Although AG learning hardly provides a true model for the acquisition of natural languages, it can be used as a tractable experimental model to study how people acquire without conscious efforts intuitive knowledge about complexly structured strings. Intuitive knowledge here means a tacit knowledge base that is obtained through unconscious, nonreflective, implicit cognitive processes and that can be used implicitly to make accurate well-formedness decisions about novel strings. In this sense, AG learning is a useful model for understanding the acquisition of natural languages. And a computer simulation of AG learning is a viable method to explore mechanisms underlying such implicit learning.

2. Learning a Finite State Grammar

A finite state automaton depicted in Figure 2 is defined by a 5-tuple \((Q, \Sigma, \delta, q_0, F)\) where:

1. \(Q\) is a finite set of states.
   \[ Q = \{S_0, S_1, S_2, S_3, S_4, S_5\} \]
(2) $\Sigma$ is a finite set of the alphabet.
$\Sigma = \{T, S, X, P, V\}$

(3) The transition function $\delta$ takes as arguments a state and a letter in the alphabet and returns a state. For example, if the FSA is in the state $S_0$ and then moves to $S_1$, producing the letter $T$, this is indicated by saying that $\delta(S_0, T) = S_1$.

(4) $q_0 = S_0$ is the start state.

(5) $F = \{S_5\}$ is the set of final states.

Grammatical strings are generated by starting at $S_0$ and then following any permissible sequence of arrows leading from $S_0$ to $S_5$. Because the same letter is associated with different arrows in finite state grammars like Figure 2, grammatical judgment of strings has to be formed on the basis of contextual information defined by the sequence of arrows permitted in the finite grammar.

Following Cleeremans et al. (1995), we employ the finite state grammar shown in Figure 3 to examine how well two SRN models can encode long-distance dependencies between the first letters (namely, $T$ and $P$ produced by the transition from $S_0$ to $S_1$ and from $S_0$ to $S_2$, respectively) and the last letters ($T$ and $P$ associated with the arrow leading from $S_{11}$ to $S_{13}$ and from $S_{12}$ to $S_{13}$ respectively) across the intervening finite state subgrammars shown in Figure 4. In the finite state grammar shown in Figure 3, when $T$ is chosen as the first letter in a string, the same letter $T$
has to be chosen as the last letter. Alternatively, the choice of $P$ as an initial letter requires the presence of $P$ in the final position in a string.

The two SRN models are built by using PDP++ software (Dawson et al. (2002) and O’Reilly and Munakata (2000)), which is a neural-network simulation system.
written in C++. The simulation itself consists of two processes. The first process is a training process where a network is trained to learn regularities formally represented in a finite state grammar by adjusting the weights of connections between units in different layers according to mathematically defined learning mechanisms (also called algorithms) with default parameters set by PDP++. In Elman network, backpropagation algorithm is employed to minimize the discrepancy between the desired output pattern and the actual output pattern, but a combination of error-driven and Hebbian algorithm is used in Leabra network. The update of context units in Leabra network is determined by the two parameters $f_{\text{hid}}$ and $f_{\text{prc}}$ which are set at .3 and .7 respectively in this simulation to facilitate the maintenance of contextual information instead of typical .7 and .3 (see (O’Reilly and Munakata (2000: section 6.6.1)).

In PDP++ software, FSA is defined as a ScriptEnv object which contains C-Super-Script code to generate an entire set of training strings at run-time. Strings are generated probabilistically, so that when there are two arrows leaving from the same state, either is selected randomly with a probability of 0.5, as exemplified below.

![Figure 5](image)

Such FSAs involving only evenly distributed probability of selection are called symmetrical FSAs, while FSAs involving unequally distributed probabilities are specified as nonsymmetrical.

The second process is a testing process where we check whether or not the trained network can make a correct prediction about the identity of the last letter when the automaton in Figure 3 arrives at the penultimate states $S_{11}$ and $S_{12}$. The correct prediction of the last letter depends wholly on the network’s capability to retain the information of corresponding initial letters as part of internal contextual information.

The performance of the network is evaluated by the relative activation of the
units representing letters. For example, the following simulation data show that the machine correctly predicts $P$ as the last letter.

\[ V \to P \quad 12 \to 13 \quad 0.284021 \quad 0.021625 \quad 0.000433 \quad 0.0058 \quad 0.613571 \]

$V \to P$ indicates that $V$ is the present input letter and the next letter is $P$. The associated $12 \to 13$ indicates that the machine is now at $S_{12}$ and will move to $S_{13}$ when it receives the next input letter $P$, as we see in Figure 3. Since the $P$ unit in the output layer gains the highest level of activity, 0.613571, the machine at $S_{12}$ has successfully made a correct prediction for the next letter. On the other hand, the data below show that the machine fails to make a correct prediction.

\[ V \to T \quad 11 \to 13 \quad 0.413495 \quad 0.016363 \quad 0.000377 \quad 0.007512 \quad 0.550466 \]

Although the machine predicts $P$ as the next letter, no such option is available for the machine at $S_{11}$.

### 3. Results of Simulations

In order to assess how well the training of the network can contribute to the performance, the network has been tested twice, namely after 100 epochs of learning and after 500 epochs of learning. In this simulation, one epoch process consists of 25 strings randomly generated by FSA. Symmetrical and nonsymmetrical FSAs are used to examine how differences in intervening context affect the learning of long-distance contingencies between the paired initial and final letters. In the nonsymmetrical FSA, the two intervening contexts are made slightly different as we see in Figure 6 on the next page, following Cleeremans et al. (1995). Thus the X arrow leading from $S_7$ to $S_4$ has a 0.8 probability of being selected, and the S arrow between $S_7$ and $S_{11}$ has a 0.2 probability. The nonsymmetrical 0.8 vs. 0.2 probabilities are also assigned to the two arrows leaving from $S_{10}$.

Ten networks initialized by different random seeds were used under each combination of two factors, namely training epochs (100 versus 500) and contexts (symmetrical versus nonsymmetrical), and the following simulation results were obtained.
A 2 × 2 ANOVA was conducted to evaluate the effects of long and short term training processes and symmetrical versus nonsymmetrical context. The effects were measured by the number of errors committed by the network in predicting the last letter in a string at penultimate states $S_{11}$ and $S_{12}$.

With respect to Elman network, the two-factor ANOVA indicated a significant main effect for the training epoch factor, $F(1, 36) = 24.91, p < .001$, and for the
context factor, $F(1, 36) = 4.77, p = .036$, and a significant interaction between training epochs and contexts, $F(1, 36) = 7.31, p = .01$. Because the interaction between training epochs and contexts was significant, we examined the simple main effects and found that there were no significant differences between 100 and 500 training epochs for symmetrical contexts, $F(1, 36) = 2.616, p = .115$, but there were significant differences between 100 and 500 epochs for nonsymmetrical contexts, $F(1, 36) = 29.600, p < .001$. Thus, at least after the training of 100 epochs, the additional training is only effective for the learning of nonsymmetrical contexts (see Cleeremans et al. (1995) for similar observations).

In the case of Leabra network, the ANOVA showed no significant results: for the training epoch factor, $F(1, 36) = 1.614, p = .212$, for the context factor, $F(1, 36) = .554, p = .461$, and for the interaction between training epochs and contexts, $F(1, 36) = .232, p = .633$. Thus, Leabra network shows no detectable learning ef-

![Figure 7](image-url)
fect in the case of long-distance dependencies in strings.

These differences in performance between the two SRN models are clearly seen in line graphs in Figure 7 above where the means of errors under different combinations of networks and contexts are displayed. But we must note as a caveat that these clear differences between the two SRN models are observed only under the common structural condition specified in Figure 1 and do not necessarily imply that the same level of difference is always observed under any other structural conditions.

References


