Chunking in a Connectionist Network

David S. Touretzky

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Incremental performance improvement with accumulated experience has been measured in human beings for a wide variety of cognitive, perceptual, and motor tasks (Newell, 1987). "Chunking" produces similar performance improvements in symbolic computer programs, such as the SOAR production system (Laird et al., 1987). Chunking takes place in SOAR by observing the working memory trace associated with a sequence of rule firings, and abstracting from this trace a chunk which in the future will produce the same results in a single step.

This paper presents a rule-following connectionist system that also improves its efficiency through chunking. It differs from symbolic production systems in several respects. Although connectionist networks may exhibit rule-following behavior, they do not necessarily contain explicit symbolic rules (Rumelhart & McClelland, 1986; Smolensky, 1988; Pinker & Prince, 1988). The system reported here learns its initial set of behaviors by back propagation from examples. Chunks are then created by a mechanism that observes input/output behavior as the network runs. The chunker is not told which features of the input were responsible for a particular output. In SOAR terminology, it has no access to a working memory trace.

The task the connectionist network is performing is string manipulation based on an abstract version of generative phonology. It was while working on a connectionist approach to phonology that I hypothesized chunking might play a role in the linguistic development of humans. Some speculations on the interaction between a chunker and the Language Acquisition Device appear at the end of this paper.

A Rule-Following Connectionist Network

Figure 1 shows part of a connectionist network that manipulates strings according to context-sensitive rewrite rules. The rewrite rules are an abstract version of classical generative phonology rules, and are shown here using classical notation. Rule R1 below says "change C to E in environments where it precedes a D." Similarly, rule R2 says "change A to B when it precedes an E."

```
R1:  C  -->  E  /  _  D
R2:  A  -->  B  /  _  E
```

Application of R1 to the string ABCD yields ABED. Figure 1 shows how this is accomplished. The input buffer, rule module, and change buffer form a three-layer feed-forward network. Symbols are sequentially shifted into the input buffer. Rule units read the buffer state and generate an output pattern in the change buffer describing the changes that are to be made to the input. (Each input buffer segment has a corresponding change buffer segment.) Three types of changes are possible: *mutation* of input tokens, *deletion* of input tokens, and *insertion* of new tokens. A String Editing Network, not shown, reads the input and change buffer patterns and generates an updated input pattern in which the specified changes have been made. The design of the String Editing Network is explained in (Touretzky, 1989).

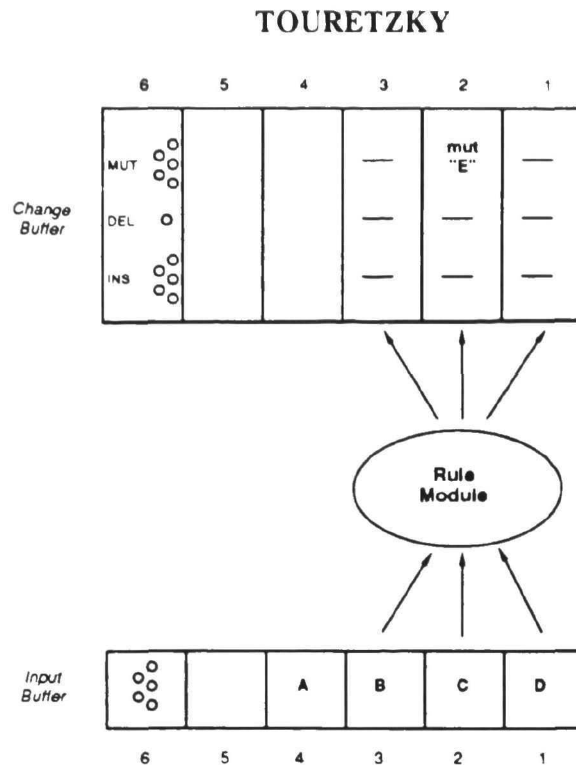

Figure 1: Part of a connectionist network for applying rewrite rules to strings.

The symbols from which strings are composed are binary feature vectors. The experiment reported here uses a representation with five "phonetic" features organized as one group of two features and one of three features. (In real phonology there are many more features; they encode the place and manner of articulation of sounds.) Features within a group are mutually exclusive. There are a total of six legal symbols, labeled A through F. The change buffer patterns use an eleven-element code for each segment: one for signaling deletion, five for describing a mutation, and five for specifying an insertion. Symbols are always inserted to the right of the corresponding input buffer segment.

Change buffer patterns are tri-state: 0 means "no change," +1 means "turn the corresponding bit in the input buffer on," and -1 means "turn the corresponding bit off." For deletion and insertion operations, -1 is treated like zero. The use of tri-state patterns causes the change buffer units to adopt the "no change" case as the default in the absence of input. Tri-state outputs are obtained using the symmetric sigmoid activation function $\sigma(x) = 2/[1 + \exp(-x)] - 1$.

The initial rules are installed by applying backpropagation to a training set of input pattern/change pattern pairs. The rule module serves as the hidden layer during learning. Once the initial rule set has been acquired, there is no supervised learning in the model. To acquire chunks, sequences of typical inputs are run through the input buffer. As it applies its rewrite rules, the model formulates chunks when two rules fire in succession, and trains itself using backprop to predict a chunked action in the appropriate context. Chunking may therefore be regarded as "self-supervised" learning, since the model is serving as its own teacher. The chunking mechanism is explained further after the next section.

Position-Independent Rules

Rules are always learned in "standard position," where the rightmost element of the rule's environment is

the rightmost element of the input buffer. However, downstream feeding relationships may require rules to apply in other positions. Consider what happens when the string ACD is shifted into the input buffer one segment at a time. The network does nothing with the initial substrings A and AC. After shifting in the D, ACD is converted to AED by rule R1. R2 should then apply to produce BED, but the AE environment for R2 is not aligned with the right edge of the input buffer; it is one segment downstream.

To allow rules to apply independent of position, we make several downstream copies of the primary rule module and constrain the link weights in each copy to be equal to the corresponding primary module weights, as shown in Figure 2. This way rules need only be learned in standard position, but they can apply anywhere they are needed. The reason for using a change description as the output representation should now be clear: the outputs of all the rule modules can be superimposed by addition at the change buffer units. If each rule module were to directly map the input string to an updated string, the outputs of multiple rule modules could not be combined.

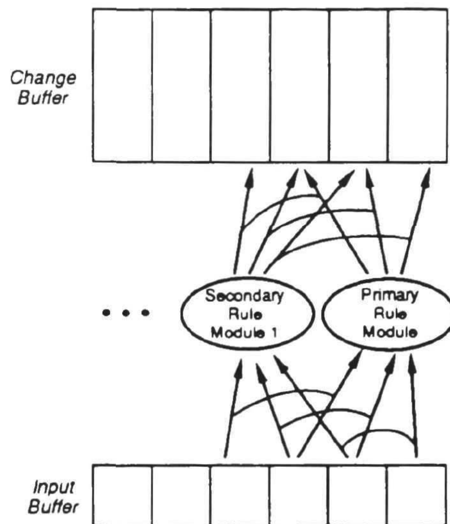

Figure 2: Link-equality constraints cause secondary rule modules to replicate the behavior of the primary module at various positions downstream.

The Chunking Mechanism

Figure 3 shows how chunking is accomplished. The model has two change buffers. The α connections, which control the Current Change Buffer, are created by back propagation learning on an initial training set supplied by the teacher. The β connections control the Chunked Change Buffer, which the network uses to teach itself new chunks.

Chunking occurs continuously as the network processes patterns flowing through its input buffer. Each time a symbol is shifted into the input buffer and a forward pass is performed, the α connections produce a Current Change Buffer pattern. If the pattern is all zeros, meaning no α rule fired, the β connections are taught to produce the same result. If the pattern is non-zero, meaning some α rule did fire, the chunker makes a note of the change buffer pattern, and the string editing network makes the requested change

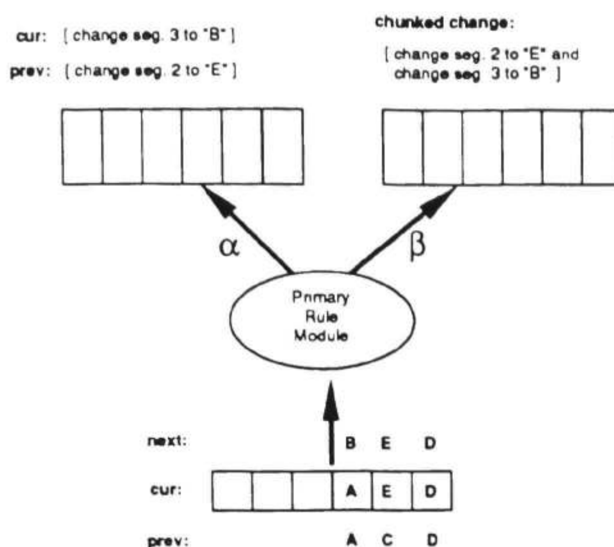

Figure 3: Chunking of rules R1 and R2 by training β connections to produce the composition of the two rules' change buffer patterns.

and updates the input buffer. After a second forward pass, if no more α rules fire, there is no sequence to be chunked. In this case the β connections are taught to imitate the change pattern produced by the single α rule. If an α rule does fire on the second forward pass, a chunk can be composed from the remembered change buffer pattern of the first rule plus the change buffer pattern of the second rule. The β connections are then taught to output this composite change pattern in the context that caused the first α rule to fire.

This training regimen ensures that the β rules will be an essential superset of the α rules. The only α rules not duplicated on the β side will be those that never fire in isolation, but only to feed another rule or as a result of a feeding rule. These non-essential α rules will be replaced by chunks. More commonly, chunked and unchunked versions of rules coexist on the β side.

A number of fine points in the training of the model need to be explained. In a chunking network the connections to rule and change buffer units should remain plastic. Plasticity can be lost if units are allowed to get too far out on the tails of the sigmoid, where the derivative goes to zero. Several steps are taken to prevent loss of plasticity. In standard back propagation the error signal of an output unit is defined to be the difference between the actual and desired outputs multiplied by the derivative of the output function (Rumelhart et al., 1986). In the chunking network the derivative term is omitted for output units.

In addition, weights must not be allowed to grow too large during training, as this can also hinder future learning. To keep the weights small and the rule units from getting too far out on the tails of the sigmoid, the model uses output training targets of $+0.5$ and -0.5 rather than $+1$ and -1. When updating the input buffer, any change buffer value greater than $+0.3$ is treated as $+1$, and any value less than -0.3 is treated as -1.

Although change buffer units use a symmetric sigmoid, rule units use the standard sigmoid. I conjecture

that rules may be learned more easily this way. Rule units are feature detectors, so when a feature is not present the unit's output should be zero. This is easily achieved with the standard sigmoid by supplying a substantial negative bias that can be counteracted only by an appropriate pattern of input features. With tri-state units it is not possible hold the output steady at zero over the entire set of inputs that aren't supposed to trigger a rule.

Finally, it should be noted that in order to learn the environments in which new chunks apply, rule units must modify not only their β output connections, but also their input connections. But this alters the rule unit's response to subsequent inputs, so it may interfere with the continued production of correct patterns in the Current Change Buffer. To prevent the model from leading itself astray, it is programmed to continually rehearse its α behaviors as it trains the β connections. Rehearsal is another instance of self-supervised learning. Each pattern the α units generate in the Current Change Buffer is "idealized" by treating all values greater than +0.3 as +0.5, all values less than -0.3 as -0.5, and all other values as 0. The difference between the actual α outputs and the idealized outputs generates an error signal that helps to readjust the input weights on each presentation, countering the disruptive effect training the β units has on the input weight pattern. The α and β sides of the model are thereby forced to compromise on an input weight pattern that allows each side to do its job.

Complex Rule Interactions

Composing a chunk from two mutation rules is easy: one simply inclusive-or's the change buffer patterns (using tri-state logic), giving the second rule priority in the case of a $+1/-1$ conflict. Composing chunks from other types of rules is slightly more complex. If the first rule inserts or deletes a segment, some portion of the second rule's change buffer pattern will need to be shifted to take this change into account before inclusive-oring the two together. If the second rule mutates a segment that was inserted by the first, the second rule's mutation pattern must be combined (with priority) with the first rule's insert pattern, not its mutation pattern. If the second rule deletes a segment that was inserted by the first rule, the first rule's insertion must be suppressed in the composed chunk. This can be accomplished by setting the insertion bits to zero.

In the simulation, chunked change buffer patterns were composed by a Lisp version of the above algorithm. However, it would be easy to construct a connectionist network to do the same task. The input would be the current and previous change buffer patterns; the output would be the composed change.

A limitation of this particular rewrite-rule architecture is that only one symbol can be inserted between each pair of symbols in the input buffer. Therefore one cannot chunk two rules if they both insert something at the same input position. In practice this situation does not seem to come up in segmental phonology, although there are multi-segment insertions at the morphological level.

Experimental Results

The initial chunker simulation used an input buffer of length six, and three rule modules, each of which looked at three adjacent input segments. The primary rule module was taught rules R1 and R2 by backpropagation on a small training set. (The training set consisted of some environments in which the rules should apply, plus some additional environments in which no rule should fire.) The following example shows the results of this training. R2 and then R1 applies, independently, in standard position, as the string AEFCD is shifted through the input buffer. Underscores denote null segments (all zeros.)

```
* (demo '(a e f c d))
    Shift A into input buffer:      _ _ _ _ _ A
    Shift E into input buffer:      _ _ _ _ A E
      Change due to rule firing:    _ _ _ _ B E        (rule R2)
    Shift F into input buffer:      _ _ _ B E F
    Shift C into input buffer:      _ _ B E F C
    Shift D into input buffer:      _ B E F C D
      Change due to rule firing:    _ B E F E D        (rule R1)
```

We next consider an example of downstream feeding of R2 by R1, which never occurred in the training data. Note that after the last symbol is shifted in, the input buffer changes twice. This is the condition allowing a chunk to be composed.

```
* (demo '(a c d))
    Shift A into input buffer:      _ _ _ _ _ A
    Shift C into input buffer:      _ _ _ _ A C
    Shift D into input buffer:      _ _ _ A C D
      Change due to rule firing:    _ _ _ A E D        (rule R1)
      Change due to rule firing:    _ _ _ B E D        (rule R2)
```

Running the network on sequences such as ACD allows it to learn chunks in self-supervised mode, by observing its own behavior. The chunk for turning ACD into BED consists of R1 plus a shifted version of R2, since R2 is applying one segment downstream. The rule units must learn to pay attention to the third segment of the buffer, whereas for R1 and R2 in isolation only the first two segments are important.

The result of chunking is shown below for the string ACDCD. (To actually use the learned chunks we replace the α weights with the learned β weights.) The ACD to BED portion of the example demonstrates the existence of the R1-R2 chunk; the CD to ED portion that follows demonstrates the preservation of R1 on the β side as an independent rule. Other inputs verified that R2 was also preserved.

```
* (demo '(a c d c d))
    Shift A into input buffer:      _ _ _ _ _ A
    Shift C into input buffer:      _ _ _ _ A C
    Shift D into input buffer:      _ _ _ A C D
      Change due to rule firing:    _ _ _ B E D        (chunk R1-R2)
    Shift C into input buffer:      _ _ B E D C
    Shift D into input buffer:      _ B E D C D
      Change due to rule firing:    _ B E D E D        (rule R1)
```

Additional experiments confirm that the network can chunk insertion and deletion rules as well as mutations. It can also combine a learned chunk with another rule to form a bigger chunk.

As long as the model's behavior is governed solely by the α connections, it will not be able to apply the chunks it has learned. An initial, brute-force solution to this problem is to simply copy the β weights to the α connections whenever the β training error is low enough. But such a drastic, global weight change

is admittedly unnatural. We are currently exploring more fluid ways of exchanging knowledge between the α and β sides. One scheme we have tried is to maintain running confidence levels for each side, and with each new input symbol, stochastically choose either the α or β change buffer pattern based on relative confidence values. Initially the β confidence is low. When the α side has successfully trained the β side, the network begins to execute a mix of α and β actions, including some learned chunks. As the β side in turn tries to teach new chunks to the α side, the α confidence level drops and the β rules take over until the new α chunks have been learned.

Interesting Chunking Phenomena

A number of interesting questions are raised by this work. One is the order in which larger chunks should be formed. Consider the feeding rule chain R1-R2-R3-R4. If the model builds at most one chunk before shifting a new symbol into its buffer, the chain will be chunked in the order (((R1 R2) R3) R4). This approach is compatible with the power law of practice cited by Newell. If the model builds a chunk whenever any pair of unchunked rules fire in sequence, the order of chunk creation will be ((R1 R2) (R3 R4)). It is not yet known which order is more compatible with the way the learning algorithm creates rule representations.

A second question is what representation the model will develop for rules that participate in multiple feeding chains. Consider a case where, for one class of inputs there is a chunk R1-R2-R3, and for another class a chunk R1-R4-R5. Since R1 is shared by both chunks and may also apply in isolation, the representations of the two chunks and the original rule should be similar, and will probably share units.

A related issue is the formation of variable-length chunks from self-feeding rules, such as this deletion rule:

```
R6:  E --> Ø / _ F
```

R6 applies three times in succession to the string BEEEF to derive BF. After chunking, BF should be obtained in a single rule firing. If the chunker is exposed to sequences of form $\{E\}^+F$ of varying length, it should build a collection of related chunks. The degree and nature of the overlap in representations of these chunks is worth investigating.

Finally there is the issue of variables appearing in rules. Variables serve either to narrow the domain of application of a rule (when the same variable appears twice on the left hand side), or to copy a value from one place to another (when the variable appears once on the left and at least once on the right hand side.) In phonology it is not too expensive to expand a variable-containing rule into a set of variable-free rules, because variables can take on only a few values. In more general symbol processing tasks this may not be feasible. It may be possible to teach a backpropagation network to implement rules with variables by encoding the value in the hidden layer activation pattern. Such a scheme would probably require a more complex hidden layer than in the present model.

Chunking and Language

The segmental phonology of any human language can be expressed by sequences of simple rewrite rules on strings. These rules are highly constrained, so that, for example, reversing the segments of a word is not possible in human phonology (Pinker & Prince, 1988). Another constraint is that there is no metathesis (switching) of non-adjacent segments. The regularity and degree of constraint of phonological processes is striking, and cries out for scientific explanation. The hypothesis motivating the work reported