The Recurrent Cascade-Correlation Architecture

Scott E. Fahlman

May 9, 1991
CMU-CS-91-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Recurrent Cascade-Correlation (RCC) is a recurrent version of the Cascade-Correlation learning architecture of Fahlman and Lebiere [Fahlman, 1990]. RCC can learn from examples to map a sequence of inputs into a desired sequence of outputs. New hidden units with recurrent connections are added to the network one at a time, as they are needed during training. In effect, the network builds up a finite-state machine tailored specifically for the current problem. RCC retains the advantages of Cascade-Correlation: fast learning, good generalization, automatic construction of a near-minimal multi-layered network, and the ability to learn complex behaviors through a sequence of simple lessons. The power of RCC is demonstrated on two tasks: learning a finite-state grammar from examples of legal strings, and learning to recognize characters in Morse code.

This research was sponsored in part by the National Science Foundation (Contract EET-8716324) and the Defense Advanced Research Projects Agency (Contract F33615-90-C-1465). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Govenrment or any of its agencies.

1. The Architecture

Cascade-Correlation [Fahlman, 1990] is a supervised learning architecture that builds a near-minimal multi-layer network topology in the course of training. Initially the network contains only inputs, output units, and the connections between them. This single layer of connections is trained (using the Quickprop algorithm [Fahlman, 1988]) to minimize the error. When no further improvement is seen in the level of error, the network's performance is evaluated. If the error is small enough, we stop. Otherwise we add a new hidden unit to the network in an attempt to reduce the residual error.

To create a new hidden unit, we begin with a pool of *candidate units*, each of which receives weighted connections from the network's inputs and from any hidden units already present in the net. The outputs of these candidate units are not yet connected into the active network. Multiple passes through the training set are run, and each candidate unit adjusts its incoming weights to maximize the correlation between its output and the residual error in the active net. When the correlation scores stop improving, we choose the best candidate, freeze its incoming weights, and add it to the network. This process is called "tenure." After tenure, a unit becomes a permanent new feature detector in the net. We then re-train all the weights going to the output units, including those from the new hidden unit. This process of adding a new hidden unit and re-training the output layer is repeated until the error is negligible or we give up. Since the new hidden unit receives connections from the old ones, each hidden unit effectively adds a new layer to the net. (See figure 1.)

Cascade-correlation eliminates the need for the user to guess in advance the network's size, depth, and topology. A reasonably small (though not minimal) network is built automatically. Because a hidden-unit feature detector, once built, is never altered or cannibalized, the network can be trained incrementally. A large data set can be broken up into smaller "lessons," and feature-building will be cumulative.

Cascade-Correlation learns much faster than backprop for several reasons: First only a single layer of weights is being trained at any given time. There is never any need to propagate error information backwards through the connections, and we avoid the dramatic slowdown that is typical when training backprop nets with many layers. Second, this is a "greedy" algorithm: each new unit grabs as much of the remaining error as it can. In a standard backprop net, the all the hidden units are changing at once, competing for the various jobs that must be done—a slow and sometimes unreliable process.

Cascade-correlation, like back-propagation and other feed-forward architectures, has no short-term memory in the network. The outputs at any given time are a function only of the current inputs and the network's weights. Of course, many real-world tasks require the recognition of a sequence of inputs and, in some cases, the corresponding production of a sequence of outputs.

A number of recurrent architectures have been proposed in response to this need. Perhaps the most widely used, at present, is the Elman model [Elman, 1988], which assumes that the network operates in discrete time-steps. The outputs of the network's hidden units at time t are fed back for use as additional network inputs at time-step $t + 1$. (See figure 2.) These additional inputs can be thought of as state-variables whose contents and interpretation are determined by the evolving weights of the network. In effect, the network is free to choose its own representation of past history in the course of learning.

Recurrent Cascade-Correlation (RCC) is an architecture that adds Elman-style recurrent operation to the Cascade-Correlation architecture. However, some changes were needed in order to make the two models fit together. In the original Elman architecture there is total connectivity between the state variables (previous

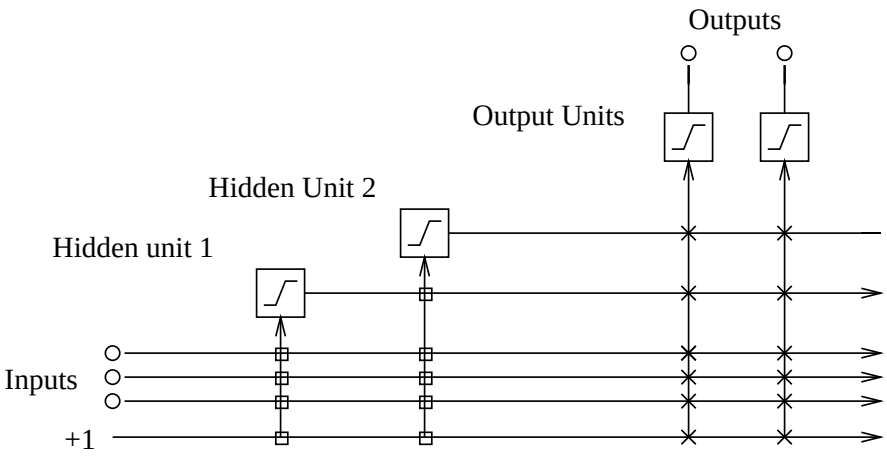

Figure 1: The Cascade-Correlation architecture after two hidden units have been added. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly.

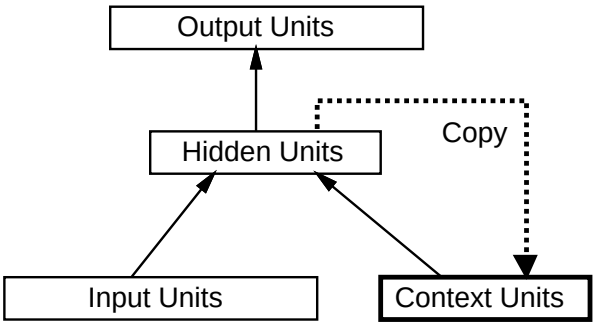

Figure 2: The recurrent network architecture of Elman.

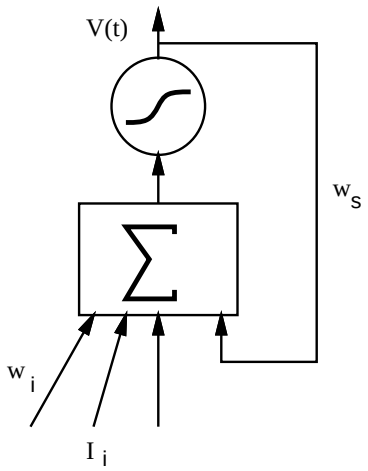

Figure 3: Candidate or hidden unit with a self-recurrent link.

outputs of hidden units) and the hidden unit layer. In Cascade-Correlation, new hidden units are added one by one, and are frozen once they are added to the network. It would violate this concept to insert the outputs from new hidden units back into existing hidden units as new inputs. On the other hand, the network must be able to form recurrent loops if it is to retain state for an indefinite time.

The solution we have adopted in RCC is to augment each candidate unit with a single weighted self-recurrent input that feeds back that unit's own output on the previous time-step (figure 3). That self-recurrent link is trained along with the unit's other input weights to maximize the correlation of the candidate with the residual error. If the recurrent link adopts a strongly positive value, the unit will function as a flip-flop, retaining its previous state unless the other inputs force it to change; if the recurrent link adopts a negative value, the unit will tend to oscillate between positive and negative outputs on each time-step unless the other inputs hold it in place; if the recurrent weight is near zero, then the unit will act as a gate of some kind. When a candidate unit is added to the active network as a new hidden unit, the self-recurrent weight is frozen, along with all the other weights. Each new hidden unit is in effect a single state variable in a finite-state machine that is built specifically for the task at hand. In this use of self-recurrent connections only, the RCC model resembles the "Focused Back-Propagation" algorithm of Mozer[Mozer, 1988].

The output, $V(t)$, of each self-recurrent unit is computed as follows:

$$V(t) = \sigma \left(\sum_i I_i(t)\, w_i \; + \; V(t-1)\, w_s \right)$$

where σ is some non-linear squashing function applied to the weighted sum of inputs I plus the self-weight, w_s, times the previous output. In the studies described here, σ is always the hyperbolic tangent or "symmetric sigmoid" function, with a range from -1 to +1. During the candidate training phase, we adjust the weights w_i and w_s for each unit so as to maximize its correlation score. This requires computing the derivative of $V(t)$ with respect to these weights:

$$\partial V(t)/\partial w_i = \sigma'(t) \;\; (I_i(t) + w_s \; \partial V(t-1)/\partial w_i)$$

$$\partial V(t)/\partial w_s = \sigma'(t)\ \left(V(t-1) + w_s\ \partial V(t-1)/\partial w_s\right)$$

The rightmost term reflects the influence of the weight in question on the unit's previous state. Since we computed $\partial V(t-1)/\partial w$ on the previous time-step, we can just save this value and use it in the current step. So the recurrent version of the learning algorithm requires us to store a single additional number for each candidate weight, plus $V(t-1)$ for each unit. At $t = 0$ we assume that the unit's previous value and previous derivatives are all zero.

As an aside, the usual formulation for Elman networks treats the hidden units' previous values as *independent* inputs, ignoring the dependence of these previous values on the weights being adjusted. In effect, the rightmost terms in the above equations are being dropped, though they are not negligible in general. This rough approximation apparently causes little trouble in practice, but it might explain the instability that some researchers have reported when Elman nets are run with aggressive second-order learning procedures such as quickprop. The Mozer algorithm does take these extra terms into account.

2. Empirical Results: Finite-State Grammar

Figure 4a shows the state-transition diagram for a simple finite-state grammar, called the Reber grammar, that has been used by other researchers to investigate learning and generalization in recurrent neural networks. To generate a "legal" string of tokens from this grammar, we begin at the left side of the graph and move from state to state, following the directed edges. When an edge is traversed, the associated letter is added to the string. Where two paths leave a single node, we choose one at random with equal probability. The resulting string always begins with a "B" and ends with an "E". Because there are loops in the graph, there is no bound on the length of the strings; the average length is about eight letters. An example of a legal string would be "BTSSXXVPSE".

Cleeremans, Servan-Schreiber, and McClelland [Cleeremans, 1989] showed that an Elman network can learn this grammar if it is shown many different strings produced by the grammar. The internal state of the network is zeroed at the start of each string. The letters in the string are then presented sequentially at the inputs of the network, with a separate input connection for each of the seven letters. The network is trained to predict the next character in the string by turning on one of the seven outputs. The output is compared to the true successor and the learning algorithm attempts to minimize the resulting errors.

When there are two legal successors from a given state, the network will never be able to do a perfect job of prediction. During training, the net will see contradictory examples, sometimes with one successor and sometimes the other. In such cases, the net will eventually learn to partially activate both legal outputs. During testing, a prediction is considered correct if the two desired outputs are the two with the largest values.

This task requires generalization in the presence of considerable noise. The rules defining the grammar are never presented—only examples of the grammar's output. Note that if the network can perform the prediction task perfectly, it can also be used to determine whether a string is a legal output of the grammar. Note also that the successor letter(s) cannot be determined from the current input alone; some memory of of the network's state or past inputs is essential.