Learning to Forget: Continual Prediction with LSTM

Felix A. Gers felix@idsia.ch
Jürgen Schmidhuber juergen@idsia.ch
Fred Cummins fred@idsia.ch
IDSIA, Corso Elvezia 36 , 6900 Lugano, Switzerland , http://www.idsia.ch/

Abstract

Long Short-Term Memory (LSTM,[5]) can solve many tasks not solvable by previous learning algorithms for recurrent neural networks (RNNs). We identify a weakness of LSTM networks processing continual input streams without explicitly marked sequence ends. Without resets, the internal state values may grow indefinitely and eventually cause the network to break down. Our remedy is an adaptive "forget gate" that enables an LSTM cell to learn to reset itself at appropriate times, thus releasing internal resources. We review an illustrative benchmark problem on which standard LSTM outperforms other RNN algorithms. All algorithms (including LSTM) fail to solve a continual version of that problem. LSTM with forget gates, however, easily solves it in an elegant way.

1 Introduction

Recurrent neural networks (RNNs) constitute a very powerful class of computational models, capable of instantiating almost arbitrary dynamics. The extent to which this potential can be exploited, is however limited by the effectiveness of the training procedure applied. Gradient-based methods ("Back-Propagation Through Time" (BPTT) [8] or "Real-Time Recurrent Learning" (RTRL) [6]) share an important limitation. The magnitude of the error signal propagated back in time depends exponentially on the magnitude of the weights. This implies that the backpropagated error quickly either vanishes or blows up [5, 1]. Hence standard RNNs fail to learn in the presence of time lags greater than 5 - 10 discrete time steps between relevant input events and target signals. The vanishing error problem casts doubt on whether standard RNNs can indeed exhibit significant practical advantages over time-window-based feedforward networks.

A recent model, *"Long Short-Term Memory"* (LSTM) [5] is not affected by this problem. LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing *constant* error flow through "constant error carrousels" (CECs) within special units, called cells. Multiplicative gate units learn to open and close access to the cells. LSTM solves complex long time lag tasks that have never been solved by previous RNN algorithms [5].

In this paper, however, we will show that even LSTM fails to learn to correctly process certain continual time series that are not *a priori* segmented into training subsequences with clearly defined ends. The problem is that a continual input stream eventually may cause the internal values of the cells to grow without bound, even if the nature of the problem suggests they should be reset occasionally. We present a remedy.

While we present a specific solution to the problem of forgetting in LSTM networks, we recognize that *any* training procedure for RNNs which is powerful enough to span long time lags must also address the issue of forgetting in short-term memory (unit activations). We know of no other current training method for RNNs which is sufficiently powerful to have encountered this problem.

2 Standard LSTM

The basic unit in the hidden layer of an LSTM network is the *memory block*, which contains one or more *memory cells* and a pair of adaptive, multiplicative gating units which gate input and output to all cells in the block. Each memory cell has at its core a recurrently self-connected linear unit called the "Constant Error Carousel" (CEC), whose activation we call the cell *state*. The CECs solve the vanishing error problem: in the absence of new input or error signals to the cell, the CEC's local error back flow remains constant, neither growing nor decaying. The CEC is protected from both forward flowing activation and backward flowing error by the input and output gates respectively. When gates are closed (activation around zero), irrelevant inputs and noise do not enter the cell, and the cell state does not perturb the remainder of the network. Fig. 1 shows a memory block with a single cell. The cell state, s_c, is updated based on its current state and three sources of input: net_c is input to the cell itself; net_{in}

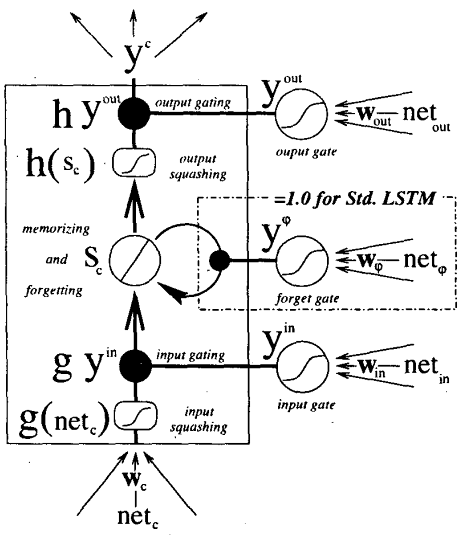

Figure 1: The LSTM cell has a linear unit with a recurrent self-connection. Input and output gates regulate read and write access to the cell whose state is denoted by s_c. The function g squashes the cell's input; h squashes the cell's output (see text for details).

and net_{out} are inputs to the input and output gates.

We consider discrete time steps $t = 1, 2, \ldots$. A single step involves the update of all units (forward pass) and the computation of error signals for all weights (backward pass). Input gate activation y^{in} and output gate activation y^{out} are computed as follows:

$$y^{out_j}(t) = f_{out_j}(\sum_m w_{out_j m}\ y^m(t\!-\!1)) \ , \quad (1)$$

$$y^{in_j}(t) = f_{in_j}(\sum_m w_{in_j m}\ y^m(t\!-\!1)) \ . \quad (2)$$

Throughout this paper j indexes memory blocks; v indexes memory cells in block j, such that c_j^v denotes the v-th cell of the j-th memory block; w_{lm} is the weight on the connection from unit m to unit l. Index m ranges over all source units, as specified by the network topology. For gates, f is a logistic sigmoid with range $[0,1]$. Net input to the cell itself is squashed by g, a centered logistic sigmoid function with range $[-2,2]$.

The internal state of memory cell $s_c(t)$ is calculated by adding the squashed, gated input to the state at the previous time step $s_c(t\!-\!1)$ $(t > 0)$:

$$s_{c_j^v}(t) = s_{c_j^v}(t\!-\!1) + y^{in_j}(t)\ g(net_{c_j^v}(t)) \ , \quad (3)$$

with $s_{c_j^v}(0) = 0$. The cell output y^c is calculated by squashing the internal state s_c via the output squashing function h, and then multiplying (gating) it by the output gate activation y^{out}:

$$y^{c_j^v}(t) = y^{out_j}(t)\ h(s_{c_j^v}(t)) \ . \quad (4)$$

h is a centered sigmoid with range $[-1,1]$.

Finally, assuming a layered network topology with a standard input layer, a hidden layer consisting of memory blocks, and a standard output layer, the equations for the output units k are:

$$y^k(t) = f_k(\sum_m w_{km}\ y^m(t\!-\!1)) \ , \quad (5)$$

where m ranges over all units feeding the output units (typically all cells in the hidden layer, the input units, but not the memory block gates). As squashing function f_k we again use the logistic sigmoid, range $[0,1]$.

Learning. See [5] for details of standard LSTM's backward pass. Essentially, as in truncated BPTT, errors arriving at net inputs of memory blocks and their gates do not get propagated back further in time, although they *do* serve to change the incoming weights. When an error signal arrives at a memory cell output, it gets scaled by the output gate and the output nonlinearity h; it then enters the memory cell's linear CEC, where it can flow back indefinitely without change (thus LSTM can bridge arbitrary time lags between input events and target signals). Only when the error escapes from the memory cell through an open input gate and the additional input nonlinearity g, does it get scaled once more and then serves to change incoming weights before being truncated.

2.1 Limits of standard LSTM

LSTM allows information to be stored across arbitrary time lags, and error signals to be carried far back in time. This potential strength, however, can contribute to a weakness in some situations: the cell states s_c often tend to grow linearly during the presentation of a time series (the nonlinear aspects of sequence processing are left to the squashing functions and the highly nonlinear gates). If we present a continuous input stream, the cell states may grow in unbounded fashion, causing saturation of the output squashing function, h. Saturation will (a) make h's derivative vanish, thus blocking incoming errors, and (b) make the cell output equal the output gate activation, that is, the entire memory cell will degenerate into an ordinary BPTT unit. This problem did not arise in the experiments reported in [5] because cell states were explicitly reset to zero before the start of each new sequence.

We tried several candidate solutions to this problem before arriving at the solution presented here. In particular, no definite advantage was found using weight decay, or by allowing internal state decay through a recurrent weight of less than unity on the CEC.

3 Solution: Forget Gates

Our solution to the above problem is to use adaptive "forget gates" which learn to reset memory blocks once their contents are out of date and hence useless. We replace standard LSTM's constant CEC weight 1.0 (Fig. 1) by the multiplicative forget gate activation y^φ.

The forget gate activation y^φ is calculated like the activations of the other gates and squashed using a logistic sigmoid, [0, 1]:

$$y^{\varphi_j}(t) = f_{\varphi_j}(\sum_m w_{\varphi_j m}\, y^m(t-1)) \,. \qquad (6)$$

y^{φ_j} functions as the weight of the self recurrent connection of the internal state s_c in equation (3). The revised update equation for s_c in the extended LSTM algorithm is (for $t > 0$):

$$\begin{aligned} s_{c_j^v}(t) &= y^{\varphi_j}(t)\, s_{c_j^v}(t-1) \\ &\quad + y^{in_j}(t)\, g(net_{c_j^v}(t)) \,, \end{aligned} \qquad (7)$$

with $s_{c_j^v}(0) = 0$. Bias weights for LSTM gates are initialized with negative values for input and output gates (see [5] for details), positive values for forget gates. This implies that in the beginning of the training phase the forget gate activation will be almost 1.0, and the entire cell will behave like a standard LSTM cell. It will not explicitly forget anything until it has learned to forget.

3.1 Backward Pass

LSTM's backward pass (see [4, 5] for details) is an efficient fusion of slightly modified, truncated BPTT, and a customized version of RTRL.

The squared error objective function based on targets t^k is:

$$E(t) = \frac{1}{2}\sum_k e_k(t)^2 \;;\; e_k(t) := t^k(t) - y^k(t) \,.$$

We minimize E via gradient descent weight changes Δw_{lm} using learning rate α:

$$\Delta w_{lm}(t) = \alpha\, \delta_l(t)\, y^m(t-1) \;,$$

where $\delta_l(t)$ for the output units is: $\delta_k(t) = f_k'(net_k(t))\, e_k(t)$ and for the output gate of j-th memory block is:

$$\begin{aligned} \delta_{out_j}(t) &= f'_{out_j}(net_{out_j}(t)) \\ &\quad \cdot \left(\sum_{v=1}^{S_j} h(s_{c_j^v}(t)) \sum_k w_{kc_j^v}\, \delta_k(t)\right) \end{aligned}$$

For weights to cell, input gate and forget gate we adopt an RTRL-oriented perspective: We define the internal state error $e_{s_{c_j^v}}$ and the partial $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ of $s_{c_j^v}$ with respect to weights w_{lm} feeding the cell c_j^v ($l = c_j^v$) or the block's input gate ($l = in$) or the block's forget gate ($l = \varphi$). For each cell, $e_{s_{c_j^v}}$ is given by:

$$e_{s_{c_j^v}}(t) = y^{out_j}(t)\, h'(s_{c_j^v}(t)) \left(\sum_k w_{kc_j^v}\, \delta_k(t)\right)$$

The partials $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ for $l \in \{\varphi, in, c_j^v\}$ are zero for $t = 0$. For $t > 0$ we get:

$$\begin{aligned} \frac{\partial s_{c_j^v}(t)}{\partial w_{l_j m}} &= \frac{\partial s_{c_j^v}(t-1)}{\partial w_{l_j m}}\, y^{\varphi_j}(t) + \Theta_l(t)\, y^m(t-1) \\ \Theta_{c_j^v}(t) &= g'(net_{c_j^v}(t))\, y^{in_j}(t) \,, \\ \Theta_{in}(t) &= g(net_{c_j^v}(t))\, f'_{in_j}(net_{in_j}(t)) \,, \\ \Theta_\varphi(t) &= s_{c_j^v}(t-1)\, f'_{\varphi_j}(net_{\varphi_j}(t)) \,. \end{aligned}$$

Updates of weights to the cell $\Delta w_{c_j^v m}$ only depend on the partials of this cell's own state:

$$\Delta w_{c_j^v m}(t) = \alpha\, e_{s_{c_j^v}}(t)\, \frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \,.$$

To update the weights of the input gate ($l = in$) and of the forget gate ($l = \varphi$), however, we have to sum over the contributions of all cells in the block:

$$\Delta w_{lm}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\, \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \,.$$

The storage complexity for the backward pass does not depend on the length of the input sequence. Like standard LSTM, extended LSTM is local in space and time.

4 Experiments

To generate an infinite input stream we extend the well-known "embedded Reber grammar" (ERG) benchmark problem [7, 2, 3, 5]. See Fig. 2. An ERG string is

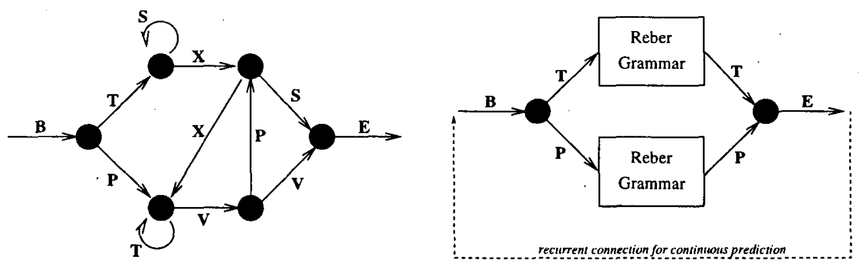

Figure 2: Transition diagrams for standard (left) and embedded (right) Reber grammars. The dashed line indicates the continual variant.

generated by starting at the leftmost node of the ERG graph, and sequentially generating finite symbol strings (beginning with the empty string) by following edges, and appending the associated symbols to the current string until the rightmost node is reached. If there is a choice, edges are chosen randomly ($p = 0.5$).

Symbols are represented using a localist encoding. The task is to read strings, one symbol at a time, and to continually predict the next possible symbol(s). While input vectors have exactly one nonzero component, target vectors may have two, because sometimes there is a choice of two possible symbols at the next step. A prediction is considered correct if the mean squared error at each of the 7 output units is below 0.49 (error signals occur at every time step).

To correctly predict the penultimate symbol (**T** or **P**) in an ERG string, the network has to remember the second symbol (also **T** or **P**) without confusing it with identical symbols encountered later. The minimal time lag is 7 (at the limit of what standard recurrent networks can manage); time lags have no upper bound though. The expected length of a string generated by an ERG is 11.5 symbols. The length of the longest string in a set of N non-identical strings is proportional to $\log N$ [4]. For the training and test sets used in our experiments, the expected value of the longest string is greater than 50.

Table 1 summarizes performance of previous RNNs (RTRL [7], "Elman net trained by Elman's procedure" [2], "Recurrent Cascade-Correlation" [3] and LSTM [5]) on the standard ERG problem (testing involved a test set of 256 ERG test strings). Only LSTM always learns to solve the task. Even ignoring the unsuccessful trials of the other approaches, LSTM learns much faster.

CERG. Our more difficult continual variant of the ERG problem (CERG) does not provide information about the beginnings and ends of symbol strings. Without intermediate resets, the network is required to learn, in an on-line fashion, from input streams consisting of concatenated ERG strings. Input streams are stopped as soon as the network makes an incorrect prediction or the 10^5-th successive symbol has occurred. Learning and testing alternate: after each training stream we freeze the weights and feed 10 test streams. Our performance measure is the average test stream size; 100,000 corresponds to a so-called "perfect" solution (10^6 successive correct predictions).

4.1 Network Topology

The 7 input units are fully connected to a hidden layer consisting of 4 memory blocks with 2 cells each (8 cells and 12 gates in total). The cell outputs are fully connected to the cell inputs, to all gates, and to the 7 output units. The output units have additional "shortcut" connections to the input units (see Figure. 3). All gates and output units are biased. Bias weights to in- and output gates are initialized blockwise: -0.5 for the first block, -1.0 for the second, -1.5 for the third, and so forth. In this manner, cell states are initially close to zero, and, as training progresses, the biases become progressively less negative, allowing the serial activation of cells as active participants in the network computation. Forget gates are initialized with symmetric positive values: $+0.5$ for the first block, $+1$ for the second block, etc. Precise bias initialization does not appear to be critical though. All other weights including the output bias are initialized randomly in the range $[-0.2, 0.2]$. There are 424 adjustable weights, which is comparable to the number used by LSTM in solving the ERG (see Table 1).

Weight changes are made after each input symbol presentation. The learning rate α is initialized with 0.5. It either remains fixed or decays by a fraction of 0.99 per time step.

Algorithm	# hidden	#weights	learning rate	% of success	success after
RTRL	3	≈ 170	0.05	"some fraction"	173,000
RTRL	12	≈ 494	0.1	"some fraction"	25,000
ELM	15	≈ 435		0	>200,000
RCC	7-9	≈ 119-198		50	182,000
Std.LSTM	3bl.,size 2	276	0.5	100	8,440

Table 1: Standard embedded Reber grammar (ERG): percentage of successful trials and number of sequence presentations until success. Weight numbers in the first 4 rows are estimates.

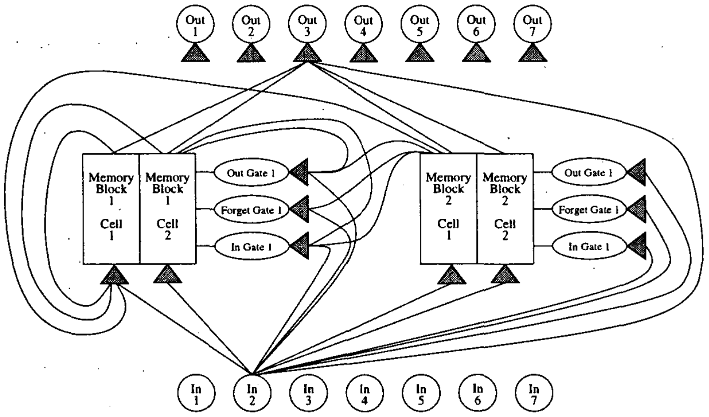

Figure 3: Three layer LSTM topology with recurrence limited to the hidden layer consisting of four extended LSTM memory blocks (only two shown) with two cells each. Only a limited subset of connections are shown.

Algorithm	%sol	train seqs
Std LSTM with external reset	74	7441
Std LSTM	0	> 30000
LSTM with Forget Gates	18	18889
LSTM with Forget Gates and sequential α-decay	62	14087

Table 2: Continual Embedded Reber Grammar (CERG): Percentage of "perfect" solutions (correct prediction of 10 streams of 100,000 symbols each), and the number of sequences presented until solution was reached.

4.2 Results

Training was stopped after at most 30000 training streams, each of which was ended when the first prediction error or the 100000th successive input symbol occurred. Table 2 compares extended LSTM with and without learning rate decay to standard LSTM. External resets (non-continual) case allow LSTM to find excellent solutions in 74% of the trials, according to our stringent testing criterion. Standard LSTM fails, however, in the continual case. Extended LSTM with forget gates can solve the continual problem. The best results were obtained by combining forget gates with learning rate decay.

4.3 Analysis of the Results

How does extended LSTM solve the task on which standard LSTM fails? Section 2.1 already mentioned LSTM's problem of uncontrolled growth of the internal states. Without forget gates, the internal states tend to grow linearly. Fig. 4 shows the evolution of the internal states s_c during the presentation of a test stream. At the starts of successive ERG strings, the network is in an increasingly active state. At some point, the high level of state activation leads to saturation of the cell outputs, and performance breaks down. Extended LSTM, however, learns to use the forget gates for resetting its state when necessary. Fig. 5 (top) shows a typical internal state evolution after learning. We see that the third memory block resets its cells in synchrony with the starts of ERG strings. The internal states oscillate around zero; they never drift out of bounds as with standard LSTM. It also becomes clear how the relevant information is stored: the second cell of the third block stays negative while the symbol **P** has to be stored, whereas a **T** is represented by a positive value. The third block's forget gate activations are plotted in Figure 5 (bottom). Most of the time they are equal to 1.0, thus letting the memory cells retain their internal values. At the end of an ERG string the

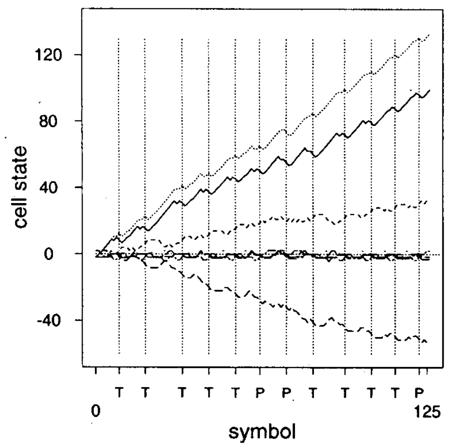

Figure 4: Evolution of standard LSTM's internal states s_c during presentation of a test stream stopped at first prediction failure. Starts of new ERG strings are indicated by vertical lines labeled by the symbols (**P** or **T**) to be stored until the next string start.

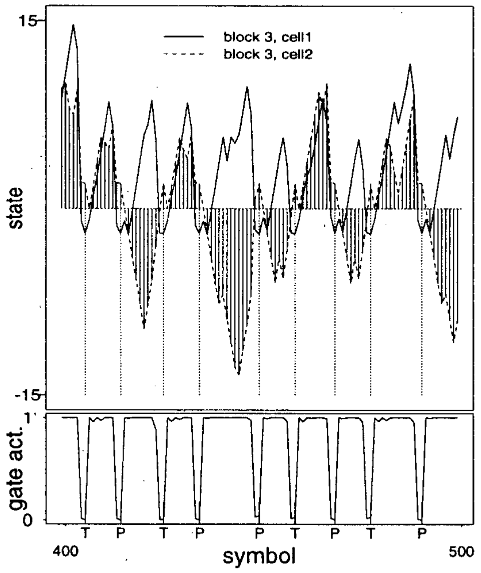

Figure 5: Top: Internal states s_c of the two cells of the self-resetting third memory block in an extended LSTM network during a test stream presentation. The figure shows 100 successive symbols taken from the longer sequence presented to a network that learned the CERG. Starts of new ERG strings are indicated by vertical lines labeled by the symbols (**P** or **T**) to be stored until the next string start. Bottom: simultaneous forget gate activations of the same memory block.

forget gate's activation goes to zero, thus resetting cell states to zero.

Analyzing the behavior of the other memory blocks, we find that only the third is directly responsible for bridging ERG's longest time lag. Other blocks reset themselves more often, presumably in order to capture the short time lag structure of the CERG strings.

5 Conclusion

Continuous input streams generally require occasional resets of the network. Partial resets are also desirable for tasks with hierarchical decomposition. For instance, reoccurring subtasks should be solved by the same network module, which should be reset once the subtask is solved. Since typical real-world input streams are not *a priori* decomposed into training subsequences, and since typical sequential tasks are not *a priori* decomposed into appropriate subproblems, RNNs should be able to *learn* to achieve appropriate decompositions. Our novel forget gates naturally permit LSTM to learn local self-resets of memory contents that have become irrelevant.

Acknowledgment. This work was supported by SNF grant 2100-49'144.96 "Long Short-Term Memory" to J. Schmidhuber.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[2] A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland. Finite-state automata and simple recurrent networks. *Neural Computation*, 1:372–381, 1989.

[3] S. E. Fahlman. The recurrent cascade-correlation learning algorithm. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *NIPS 3*, pages 190–196. San Mateo, CA: Morgan Kaufmann, 1991.

[4] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. Technical Report IDSIA-01-99, IDSIA, Lugano, CH, 1999.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[6] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

[7] A. W. Smith and D. Zipser. Learning sequential structures with the real-time recurrent learning algorithm. *International Journal of Neural Systems*, 1(2):125–131, 1989.

[8] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Back-propagation: Theory, Architectures and Applications*. Hillsdale, NJ: Erlbaum, 1992.