Report Number: WUCS-93-26

1993-01-01

TRAINREC: A System for Training Feedforward & Simple Recurrent Networks Efficiently and Correctly

Barry L. Kalman and Stan C. Kwasny

TRAINREC is a system for training feedforward and recurrent neural networks that incorporates several ideas. It uses the conjugate-gradient method which is demonstrably more efficient than traditional backward error propagation. We assume epoch-based training and derive a new error function having several desirable properties absent from the traditional sum-of-squared-error function. We argue for skip (shortcut) connections where appropriate and the preference for a sigmoidal yielding values over the [-1,1] interval. The input feature space is often over-analyzed, but by using singular value decomposition, input patterns can be conditioned for better learning often with a reduced number of input units.... Read complete abstract on page 2.

TRAINREC: A System for Training Feedforward & Simple Recurrent Networks Efficiently and Correctly

Barry L. Kalman and Stan C. Kwasny

Complete Abstract:

TRAINREC is a system for training feedforward and recurrent neural networks that incorporates several ideas. It uses the conjugate-gradient method which is demonstrably more efficient than traditional backward error propagation. We assume epoch-based training and derive a new error function having several desirable properties absent from the traditional sum-of-squared-error function. We argue for skip (shortcut) connections where appropriate and the preference for a sigmoidal yielding values over the [-1,1] interval. The input feature space is often over-analyzed, but by using singular value decomposition, input patterns can be conditioned for better learning often with a reduced number of input units. Recurrent networks, in their most general form, require special handling and cannot be simply a re-wiring of the architecture without a corresponding revision of the derivative calculations. There is a careful balance required among the network architeucture (specifically, hidden and feedback units), the amount of training applied, and the ability of the network to generalize. These issues often hinge on selecting the proper stopping criterion. Discovering methods that work in theory as well as in practice is difficult and we have spent a substantial amount of effort evaluating and testing these ideas on real problems to determine their value. This paper encapsulates a number of such ideas ranging from those motivated by a desire for efficiency of training to those motivated by correctness and accuracy of the result. While this paper is intended to be self-contained, several references are provided to other work upon which many of our claims are based.

TRAINREC:
A System for Training Feedforward & Simple Recurrent Networks Efficiently and Correctly

Barry L. Kalman and Stan C. Kwasny

WUCS-93-26

May 1993

This material is based upon work supported by the National Science Foundation under Grant No. IRI-920198

TRAINREC:

A System for Training

Feedforward & Simple Recurrent Networks

Efficiently and Correctly

Barry L. Kalman and Stan C. Kwasny
Department of Computer Science
Washington University
St. Louis, MO 63130

(314) 935-7539
(314) 935-6123

barry@cs.wustl.edu
sck@cs.wustl.edu

This material is based upon work supported by the National Science Foundation under Grant No. IRI-9201987.

ABSTRACT

TRAINREC is a system for training feedforward and recurrent neural networks that incorporates several ideas. It uses the conjugate-gradient method which is demonstrably more efficient than traditional backward error propagation. We assume epoch-based training and derive a new error function having several desirable properties absent from the traditional sum-of-squared-error function. We argue for skip (shortcut) connections where appropriate and the preference for a sigmoidal yielding values over the [-1,1] interval. The input feature space is often over-analyzed, but by using singular value decomposition, input patterns can be conditioned for better learning often with a reduced number of input units. Recurrent networks, in their most general form, require special handling and cannot be simply a re-wiring of the architecture without a corresponding revision of the derivative calculations. There is a careful balance required among the network architecture (specifically, hidden and feedback units), the amount of training applied, and the ability of the network to generalize. These issues often hinge on selecting the proper stopping criterion.

Discovering methods that work in theory as well as in practice is difficult and we have spent a substantial amount of effort evaluating and testing these ideas on real problems to determine their value. This paper encapsulates a number of such ideas ranging from those motivated by a desire for efficiency of training to those motivated by correctness and accuracy of the result. While this paper is intended to be self-contained, several references are provided to other work upon which many of our claims are based.

1.0 Introduction

The popularity of neural networks has increased recently as researchers have realized the importance of recurrent networks. Recurrent architectures have the advantage of being able to remember, to a some degree, what inputs or events have occurred earlier in a sequence and are able to exert control of future decision-making based on these past events. The challenges of recurrent networks are similar but not precisely the same as those for ordinary, feedforward networks.

Neural networks, as a field, is graduating from the novelty and toy problem stage and is becoming an important and useful tool for a wide variety of problem areas. As our understanding of the methodology increases through research, we become better equipped to address tasks on the scale of practical, real-world problems. Likewise, it seems, as our ability to train larger and more powerful networks increases, we find more useful activities for those networks. Research contributions resulting from the study of biological systems, while extremely valuable in finding new research directions, place additional demands and burdens on the task of training. Thus, efficient training methods are essential for the field to progress.

In this article we present a unique collection of methods and features which we have combined into an efficient system called TRAINREC, for training feedforward neural networks (NNs), simple recurrent neural networks (SRNs) and recursive auto associative memories (RAAMs). The emphasis in TRAINREC is on training SRNs and RAAMs and hence the name. We have established the effectiveness of TRAINREC by applying it to dozens of problems.

Our approach in this paper is to examine backward error propagation (backprop) under popular assumptions (e.g., those contained within the software package by McClelland & Rumelhart, 1988) and point out where they can be improved or altered slightly to achieve a more positive result. We also point out areas of popular misconceptions where appropriate. We begin by stating some of our motivations.

1.1 Motivations

As the problems we were addressing grew beyond the limitations inherent in the backprop algorithm, we sought to find improved techniques for choosing a network architecture and accelerating the convergence properties of our networks. We became interested in the whole gamut of improvements from connectivity issues to the choice of error function to determining the correct choice of feature space. Over the past several years, we have developed TRAINREC to embody those discoveries and techniques.

In this subsection we present some of the assumptions, techniques and features that are usually presented as *the standard way* to construct and train all varieties of neural networks. By carefully analyzing these assumptions, we have managed to introduce improvements in virtually every aspect of it. Below we present a variety of standard features together with some comments on what direction to take to improve them.

- **The [0, 1] interval.** The [-1, 1] interval is better for Boolean functions and many other problems.
- **The sum of squares error function.** Sum of squares was designed for infinite intervals and most neural network problems operate over a finite output interval.
- **Layered connectivity.** Most networks use completely connected layers of units with each layer completely connected to the preceding layer. We routinely use skip connections (also called shortcut connections) to completely connect each layer with *every* preceding layer. Most inputs to a neural network have a significant linear component. Without skip connections, the burden is on the hidden layer(s) to learn both linear and non-linear features of the data. This creates difficulties which skip connections overcome by permitting the linear (or perceptron) part of the mapping to be associated with the direct connections from input to output layers. See Figure 1.

FIGURE 1 ABOUT HERE

- **Perfect match decision criterion.** The idea that the correct output is selected only if the output vector is within a small tolerance of the target vector is too restrictive for categorization problems.
- **Mean square error stopping criterion.** The network can be very wrong on a few important unit activations even when the mean square error is very small. This can lead to poor training especially for categorization problems.
- **Intense feature analysis.** Much time is wasted in deciding the perfect set of features and values for them when the input set is constructed. We show how features can be analyzed and reduced through singular value decomposition.
- **Pattern based training.** For all the problems we have worked on, particularly ones using recurrent networks, epoch-based training is far more efficient than pattern-based training. We predicate our analysis on the assumption that epoch-based training can be done.
- **Linear convergence of backprop.** For historical reasons backprop has become highly popular. It is a linearly convergent algorithm and does not take into account error function features for less-constrained, recurrent architectures. We have chosen to utilize the conjugate gradient method, a superlinear convergence technique.
- **Overly constrained architectures for SRN.** Because of backprop, most SRNs are designed with only self loop feedback. This means that the forward connection from a context node only exists with the hidden node that spawned it. More general architectures are desired.
- **Static targets for Recursive Auto-Associative Memories.** RAAMs cleverly use auto-associativity to permit representation of complex structures as distributed patterns (see [P90] and [KK93]). In our experience, the effect of the variability of the target is usually not properly considered under standard implementations of backprop.

1.2 Primary Discoveries

In building TRAINREC we made several discoveries which are helping us improve the efficiency of training. While individually each discovery contributes modestly toward these improvements, taken collectively they combine into an extremely effective training tool. Often, successful training is impossible, in our experience, without this combination. Here, we briefly introduce the primary discoveries. These are later discussed in more detail.

- **A new error function.** In choosing to use the conjugate gradient algorithm, we found that it worked poorly with the sum of squares error function. The cross entropy error function also had serious difficulties at the ends of the output interval. The Kalman-Kwasny error function has been a key element in the success of this effort.
- **Use of singular value decomposition to preprocess input.** We have found it productive to use SVD on the collection of input patterns. The benefits are two-fold: (i) re-orientation of the input space so that inputs are orthogonally aligned; (ii) analysis of input units for the purpose of eliminating useless ones.
- **The bottleneck of SRN and sequential RAAM training.** Through algorithmic complexity analysis we found that evaluating derivatives is the major bottleneck for training this type of network. By reducing the frequency of derivative computations, we improve overall efficiency greatly.
- **Use of settling and voting.** In training a SRN for tasks involving noisy data (for example, our work on identifying which of two languages is being spoken [KKEW92] and [WKKE93]) we found that allowing the network to "settle" over a prefix of patterns is very helpful. We also found that a voting scheme based on the behavior of the cumulative binomial distribution can bootstrap a weakly performing network into an asymptotically perfect separator.

1.3 Overview

The remainder of the article contains four sections. In section 2 we discuss the methods which improve efficiency without affecting the parameters of the network. These methods are the Kalman-Kwasny error function, the use of skip connections, the use of singular value decomposition to preprocess inputs and the use of vector best match as a correctness criterion. In section 3 we describe the methods which increase efficiency during training. These are the conjugate gradient algorithm and derivative free line search with adaptive step size control. We also emphasize methods for terminating training with maximum generalization. In section 4 we present special techniques we have used for specific network architectures and specific problems. We show how to properly evaluate derivatives for SRNs and RAAMs. We discuss the use of settling and voting to bootstrap performance of our network used in language identification. We show the importance of counting states that emerge while training SRNs and RAAMs.

2.0 Statics

In this section we discuss techniques which do not depend directly upon the network parameters, such as size, feedback, etc. These are the Kalman-Kwasny error function, skip connections, singular value decomposition of the inputs and use of vector best match as a correctness criterion.

2.1 The Kalman-Kwasny Error Function

We first presented the Kalman-Kwasny error function in [KK91] and we gave its derivation along with the importance of the hyperbolic tangent sigmoidal in [KK92]. Here we summarize those ideas based on [KK92]. First we present the desirable behaviors of an error function on a finite interval. Next we show the derivation of the error function. Finally, we review how to scale the sigmoidal function for maximum fairness.

Given our assumption of epoch-based training, we can analyze the characteristics of the error function over an entire collection of patterns. From our studies on the convergence of the conjugate gradient method we have identified four criteria for the behavior of an error function on a finite interval. Given the interval [-1,1], a generic error function can be written as:

$$\Phi = \sum_{p} \sum_{k} g_{pk} \qquad \text{(EQ 1)}$$

where p is a pattern, k identifies an output unit, g_{pk} is a function of the target value t_{pk} and the activation value a_{pk} and represents the quantity of error resulting at unit k for pattern p. How should g_{pk} behave?

The four behavioral criteria we desire of g are:

(1) Behavior when the activation approaches the target:

$$\lim_{a \to t} g = 0 \qquad \text{(EQ 2)}$$

(2) Behavior when the activation approaches the end of the interval furthest from the target, where the end value is given by c and the constant c is -1 when the target is positive and +1 when the target is negative:

$$\lim_{a \to c} g = \infty \qquad \text{(EQ 3)}$$

(3) Behavior of the derivative when the activation approaches the target:

$$\lim_{a \to t} g' = 0 \qquad \text{(EQ 4)}$$

(4) Behavior of derivative when the activation approaches the other end of the interval:

$$\lim_{a \to c} |g'| = \infty \qquad \text{(EQ 5)}$$

Often, g_{pk} is chosen as the square of the error, e^2_{pk}, where $e_{pk} = (t_{pk} - a_{pk})$, but in that case, only properties EQ 2 and EQ 4 are satisfied. The other two properties are needed to discourage exactly opposite values which lead to local extrema and exterior saddle points. In order to satisfy all four properties, we have been investigating a family of scaled error functions of the form $g_{pk} = \frac{e^2{}_{pk}}{s_{pk}}$ under the conditions above. But determining a suitable error function g depends not only on this behavior, but also on the sigmoidal squashing function σ, which is used to compute the activation value of the output unit from the activations of the units to which it is connected. The amount of excitation is given by:

$$x_{pk} = \sum_i w_{ki} a_{pi} + b_k \qquad \text{(EQ 6)}$$

where the w_{ki} are the weight parameters on the connections and b_k is the bias term of the output unit. The activation is then computed by:

$$a_{pk} = \sigma(x_{pk}) \qquad \text{(EQ 7)}$$

If the scaling factor, s_{pk}, is chosen to be $1 - \sigma^2$, then the first two behavior properties are met, while if it is chosen to be the derivative, σ', then the latter two behavior properties are met. Therefore, the four criteria and the finite interval force the following differential equation to hold:

$$\sigma' = \lambda(1 - \sigma^2) \qquad \text{(EQ 8)}$$

For the interval [-1, 1], the only functional form which satisfies EQ 8 is:

$$\sigma(x) = \tanh(\lambda x) \qquad \text{(EQ 9)}$$

If $e_{pk} = (t_{pk} - a_{pk})$, this relation leads to the error function:

$$g_{pk} = \frac{e_{pk}^2}{1 - a_{pk}^2} \qquad \text{(EQ 10)}$$

We have shown in [KK92] that selecting the value:

$$\lambda = 1.5 \qquad \text{(EQ 11)}$$

maintains an equitable scaling of weight layers for training purposes.

In [KK91] we reported on the performance of our error function on the problem of training a feedforward network to be a deterministic parser (see Marcus [M80]) for a medium sized subset of English grammar. The network had 53 inputs, 22 outputs, 30 hidden units and no skip connections. The data set consisted of 113 training patterns. Table 1 contains a summary of results from [KK91]. These results are strong evidence for using the Kalman-Kwasny error function.

TABLE 1. Results for Medium Grammar Problem

Error Function	Optimization Method	Epochs	Presentations	CPU Time(sec)
Sum of Squares	Backprop	∞	∞	∞
Kalman-Kwasny	Backprop	13000	146900	230185
Sum of Squares	Conjugate Gradient	639	72151	6001
Kalman-Kwasny	Conjugate Gradient	237	26781	2300

2.2 Using Skip Connections

Most problems have a significant linear component in their solutions. Some require no non-linear component and are therefore solvable by a perceptron. Connection of input units to output units places the linear component of the solution in those connections while the non-linear component is isolated in the hidden layer weights [LH92].

For a simple example, consider the simplest architecture capable of learning XOR. Connecting inputs to outputs is required to give a 2-1-1 network. Other evidence comes from work on genetic algorithms used to prune neural nets [WB90] which shows that connections from input to output are important. Without skip connections more nodes are needed in the hidden layer to account for the linear component of the solution. We use one hidden layer because it is the smallest number for which non-linear separation can be effected. De Villiers and Barnard found that with nearly equal numbers of weights feedforward networks with one and two hidden layers performed equally well and networks with two hidden layers had more frequent occurrences of local extrema during training [VB93]. If one is testing for the optimal number of hidden units for a network then the case of zero hidden units is a natural limiting case. Of course, TRAINREC permits disconnection of inputs and outputs by user choice. This is an essential choice, of course, for RAAM networks.

In [KKA93] we reported on several problems on which we used TRAINREC. We employed skip connections in all of these cases. Since a significant fraction of the variables of each network are for the skip connections and from the size of the hidden layers required for the networks to solve each of these problems, we conclude that each has a significant linear component. Table 2 summarizes these networks. The middle number for each network size is the number of hidden units. The percentage of the variables taken up with skip connections reflects the degree of linearity present in the problem.

TABLE 2. Skip Connection Summary

Problem	Network Size	Number of Variables	Percent in Skip Connections
Bone Tumor Classification	110-0-45	4995	99
Learning 76 Grammar Rules	51-13-76	5616	69
Language Identification From Speech	202-10-2	2456	16

For our recurrent networks we use only feedback from the hidden layer [E88]. We call the nodes to which the outputs of the hidden units are fed back the *pseudo input layer*. We connect the pseudo input layer just as we do the ordinary input layer. This is illustrated in Figure 2. Others [KH91] use only self feedback in which feedback units are only connected to themselves. In section 4 we show how the pseudo input can be treated as ordinary input.

FIGURE 2 ABOUT HERE

2.3 Singular Value Decomposition for Preprocessing Input

We discussed the use of singular value decomposition(SVD) to preprocess input in [KKA93]. We reported that the use of SVD along with an affine transformation to place the inputs in the interval [-1, 1] often allowed training to proceed when it was previously impossible. SVD may also reduce the number of input units needed to represent the input without a significant effort in feature extraction. We also reported a transformation back from the parameters of the transformed input into the parameters of the original input. We summarize those results here.

Let A be the original input matrix which is $p \times I$, where p is the number of patterns and I is the number of input units. Then the SVD of A is:

$$A = UGV^T \qquad \text{(EQ 12)}$$

where U and V are orthonormal and G is diagonal. If SVD causes h columns to be removed then let $I' = I - h$ be the number of remaining columns. U is $p \times I'$, G is $I' \times I'$ and V is $I' \times I$. Let Q = UG and construct an affine transformation that maps the transformed patterns of Q into [-1, 1] by computing:

$$r_i = \min_{1 \le k \le p} Q_{ki} \qquad \text{(EQ 13)}$$

$$s_i = \max_{1 \le k \le p} Q_{ki} \qquad \text{(EQ 14)}$$

$$c_i = \frac{2}{s_i - r_i} \qquad \text{(EQ 15)}$$

$$d_i = -\frac{c_i}{2}(s_i + r_i) \qquad \text{(EQ 16)}$$

A' is the transformed input matrix of p patterns over I' input units and:

$$A'_{ki} = Q_{ki}c_i + d_i \qquad \text{(EQ 17)}$$

The training takes place using A'. After training it is useful to transform the parameters of the network to a network which uses the original input. We show in [KKA93] that for weights, w'_{kj}, on connections which involve the input units, weights that perform equivalently on the original patterns can be obtained as follows:

$$w_{ij} = \sum_k Z_{ik} w'_{kj} \qquad \text{(EQ 18)}$$

$$b_j = \sum_k w'_{kj} d_k + b'_j \qquad \text{(EQ 19)}$$

where $Z_{ik} = V_{ik} c_k$.

Often we will want to add more patterns to train the network. We want to use the network parameters of the previous training run and SVD-affine preprocessing. If one uses the orthonormal properties of V and equations 18 and 19, it is easy to derive transformations to compute w' and b' from the SVD-affine preprocessing of the enlarged input set as follows:

$$w'_{kj} = \sum_i w_{ij} (V_{ik} / c_k) \qquad \text{(EQ 20)}$$

$$b'_j = b_j - \sum_k \sum_i (w_{ij} d_k V_{ik}) / c_k \qquad \text{(EQ 21)}$$

This technique is very important. It reduces the need for extensive feature analysis often performed to generate a useful set of features which reflect the information present in a particular domain. As long as the necessary features are present, SVD will determine sufficiency.

As an example, in an experiment using the NETTalk data set to train a recurrent network, the seven character positions required for the input buffer in the feedforward architecture used originally were limited to only four, consisting of the middle position and the three forward positions. SVD reduced the $4 \times 29 = 116$ input units to 103 input units required to train on the 1,016 words from the dictionary.

Similarly, for the bone tumor identification task mentioned in Table 2, SVD reduced the number of input units from 110 to 95. This reduced the number of variables of the network from 4995 to 4275. This size reduction can lead to a significant reduction in training time. For another problem on which we are currently working, SVD reduces the number of inputs from 38 to 34. Without the SVD-affine method TRAINREC fails to train adequately any of the three problems mentioned in Table 2.

2.4 Vector Best Match as a Correctness Criterion

For a given set of weights, there must be a way of judging the correctness of the outputs from a network. We measure correct selection of output vectors in two ways: perfect match (PM) and best vector match (VBM). Each has its own usefulness.

With PM we test if the infinity norm of the difference vector between an output vector and its target vector is less than a specified tolerance. (The infinity norm of a vector is defined as the maximum absolute value over all its components.) This is useful when all or part of the target data is distributed over the entire output unit range. This situation occurs, for example, with all distributed pattern targets.

VBM works well for problems in which one output unit is assigned for each category to be distinguished by a network. Here, we compute unit vectors for the targets and outputs. The unit target vector which forms the angle having the largest cosine with the unit output vector is the winning output vector and the one whose category is chosen as the category for the output. If this is the same as the target category, then the selection is correct, otherwise it is incorrect. This category selection is used to build the confusion matrix which is used for a χ^2 test and for a technique of maximizing worst case performance. Both of these are measures of generalization and are further discussed in Section 3.4.

VBM works especially well when one output unit is designated as a catchall for cases where the predicted category is none of the those which come from the model. This is because every vector is orthogonal to the zero vector. VBM is used quite effectively, for example, in NetTalk [SR87].

In our work on RAAMs, we have both distributed patterns and category vectors as outputs. To test which symbol is represented by the output vector that is extracted we use VBM. The choice of the empty symbol is very important in this work. If we represent it by a vector of all -1's, our VBM algorithm will convert it to the zero vector in the [0,1] interval. There is no way around this since the zero vector itself will represent some symbol. After affinely transforming it, the target vector produced by the empty symbol will be the zero vector. The scalar product of the zero vector with any vector is zero and this means anything close to and even exactly the same as the representation for the empty symbol will not be correctly categorized. A simple solution, which works well in practice, is to create a special output unit for the empty symbol which is only +1 for this symbol and -1 otherwise. The rest of the output units are always -1 for the empty symbol.

We have found that VBM very often is a better measure of learning than either PM or the value of the error function. Therefore we use it as our primary correctness predictor and should be used whenever there is doubt about which one to choose.

3.0 Dynamics

In this section we discuss techniques which speed the training of neural networks. We discuss our use of the conjugate gradient algorithm(CGA) and Brent's derivative free line search(DFLS). Since evaluation of derivatives for SRNs and RAAMs is very expensive, DFLS with the CGA provides a very efficient approach to training these networks. We also discuss an adaptive step size control (ASSC) technique which further enhances DFLS.

3.1 Conjugate Gradient Algorithm

We use the CGA because it is quadratically convergent, it behaves as well as backprop in tough regions and it is possible to use less expensive DFLS for most of the training. Others (see, for example [B92]) have also found it very useful. Because of quadratic convergence it uses fewer epochs (often many fewer) than backprop. The CGA requires linear storage (in weights) while Newton-style quadratically convergent methods require quadratic storage. CGA's generalization properties are very close to those of backprop. We originally reported on the behavior of CGA in [K90]. Code for the CGA is discussed in [P71 and PFTV88]. The results in Table 1 also give strong evidence for the use of CGA over ordinary backprop.

3.2 Derivative-Free Line Search

The CGA requires a line search algorithm to find a local minimum in a particular direction for each iteration. Two line search algorithms are Brent and Dbrent [PFTV88]. Dbrent requires the magnitude of the gradient in the search direction for each "inner" iteration. Brent only requires the value of the error function for each inner iteration.

First we analyze training for ordinary feedforward networks to see the effect of using DFLS. Let I be the number of input units, H be the number of hidden units and O be the number of output units. We assume skip connections. Let De be the time complexity of a derivative epoch in floating point operations(flops) and Fe be the same for a derivative-free epoch. Let Td and Tf be the average time complexities of a CGA iteration which uses derivative and derivative free line search, respectively.

The leading terms in the complexity expressions are by algorithmic analysis of flops:

$$Fe = 2p\,(H \times I + O \times (H + I)) \quad \text{(EQ 22)}$$

$$De = 4p\,(H \times I + O \times (H + I)) \quad \text{(EQ 23)}$$

With the Dbrent algorithm for linesearch we observe that one CGA epoch requires eight epochs on average, seven of which are used in the line search.The Brent (derivative free) requires one derivative epoch and nine derivative free on average. Hence, $\frac{De}{Fe} = 2$, $Td = 8De$ and $Tf = De + 9Fe$. Accordingly,

$$\frac{Td}{Tf} = \frac{16}{11} \quad \text{(EQ 24)}$$

One should note that every epoch in ordinary backprop is a derivative epoch. The ability of CGA to use derivative free epochs is alone enough to recommend it. With SRNs and RAAMs the improvement due to DFLS is much more noticeable.

Here we analyze SRNs where the number of feedback units is the same as the number of hidden units. The SRN result for Fe is:

$$Fe = 2p\,(H \times (I+H) + O \times (2 \times H + I)) \quad \text{(EQ 25)}$$

In section 4 we present the algorithm for computing derivatives which depend on feedback for a SRN. An analysis of that algorithm leads to:

$$De = Fe + 2pH^2\,(\,(I+H) \times (H + 2 \times O)\,) \quad \text{(EQ 26)}$$

With SRNs we have observed the same relationships for Td and Tf just shown. Asymptotically with H we get: $\frac{Td}{Tf} = 8$

It's no wonder that many researchers who use ordinary recurrent backprop either limit feedback to self loops or ignore feedback derivatives in the gradient computation and suffer the consequences. We feel that self loops are too limiting and ignoring feedback derivatives can be disastrous.

Since we use derivative free line search in our recurrent CGA we do not ignore feedback derivatives and do not limit our recurrent architecture. Because the growth of the recurrent iterations is $O(H^4)$, we are careful to limit the growth of the hidden layer.

Here we analyze RAAMs where $O = H + I$. The SRN version of RAAMs that we use is described in [KKC93]. By definition, we are not able to use skip connections with our RAAMs. We get:

$$Fe = 4pH\,(I+H) \quad \text{(EQ 27)}$$

$$De = Fe + 2pH^2 (H + I)\ (2H + I) \tag{EQ 28}$$

which leads to $\frac{Td}{Tf} = 8$, the same as for SRNs.

3.3 Adaptive Step Size Control

Line search requires a region known to contain a local minimum before it can start. The algorithm mnbrak [PFTV88] locates such a region. It requires three points along the search direction to initialize it. At the beginning of CGA we use the ratio of the error function to the magnitude of the gradient as a start for mnbrak.

During CGA we use ASSC to reduce the number of epochs required by mnbrak. This is usually about three. The mnbrak algorithm requires an initial range to construct a region guaranteed to contain a minimum in the direction of the line search. We reason that the size of this region does not vary much from one CGA iteration to the next. If the actual step along the search direction is above 0.8 of the original interval length ASSC doubles the size of the initial search range; if the actual step is below 0.2 of the original length, ASSC halves the size of the initial search range. We observe that less than 50% of the DFLS epochs are needed when compared with that required by a fixed initial search range.

3.4 Tests of Generalization

We test generalization by running one epoch of the test set through the network every k CGA iterations for $1 \leq k \leq 10$. If training causes test set performance to be very high, then a value of k = 10 works well. If poor performance is expected, a value of k = 1 is used.

To measure the performance of a network during training we have looked at several criteria. The first is overall performance. If some output categories occur much more frequently than others, a network which assigns all results to the more frequent categories very often will be correct. If predicting occurrences of the less frequent categories is essential then predicting only frequent categories will be almost useless. Suppose a particular problem has two output categories and one of them occurs in 95% of the patterns. Just by always predicting this category a network will be right 95% of the time and this will lead to a small mean-squared error term. If making the correct prediction for the smaller category is important such a network will be a failure.

The second measure is a χ^2 computation (as specified in [PFTV88]) on the confusion matrix. This measure tends to be high when performance is distributed over both frequent and infrequent categories. Unfortunately it can be high under other conditions. The confusion matrix is the same as a two-way contingency table discussed in [MGB74]. Construction of the confusion matrix is discussed in Section 2.4.

The third measure concerns maximizing worst case performance. We save the network whenever the observed lowest fraction correct in the test set categories increases. If this criterion remains constant, then we save the network if the χ^2 measure on the test set increases. If both criteria remain constant we save the network if the overall training error function decreases. Since these techniques involve measurements on a test set, we suggest providing additional patterns for the purpose of independently measuring performance.

The number of hidden units strongly effects the generalization ability of a network. We hypothesize that an upper bound for the size of the hidden layer is the minimum number of hidden nodes needed for a network that makes the right prediction for every pattern available. A larger hidden layer will no doubt lead to overtraining. A smaller hidden layer usually gives better generalization. As an upper bound, we find the minimum hidden layer that can explain all the data.

4.0 Specific Concepts

In this section, we discuss special techniques used for specific network architectures and specific problems. We show how to correctly evaluate derivatives caused by feedback in

SRNs and RAAMs. We discuss the use of settling and voting to bootstrap the performance of a network we are using for language identification. We note that identical training patterns can easily occur when considering the sequential data of the sort that occurs with SRNs and RAAMs and we discuss our experiences in addressing this issue.

4.1 Feedback Derivatives for SRNs

The algorithms we use in our training program for recurrent neural networks require careful consideration. We present the derivation of the equations used in the critical section of our code and discuss how performance might be improved through the use of supercomputer architectures.

First, we show the derivatives for a SRN. Recall that the Kalman-Kwasny error function, shown in its entirety, is:

$$\Phi = \sum_{p}\sum_{i}\frac{(t_{pi}-a_{pi})^{2}}{1-a_{pi}^{2}} \qquad \text{(EQ 29)}$$

where p runs over patterns and i over output units. Target, t_{pi}, and activation, a_{pi}, are specific to an input pattern presentation and an output unit. The derivative equations are:

$$\frac{\partial\Phi}{\partial Y} = \sum_{p}\sum_{i}\frac{2}{(1-a_{pi}^{2})^{2}}(t_{pi}-a_{pi})\left((t_{pi}-a_{pi})a_{pi}-(1-a_{pi}^{2})\right)\frac{\partial a_{pi}}{\partial Y} \qquad \text{(EQ 30)}$$

where Y represents any weight or bias of the network and

$$\frac{\partial a_{pi}}{\partial Y} = 1.5\,(1-a_{pi}^{2})\,\{\sum_{k}(w_{ik}\frac{\partial a_{pk}}{\partial Y}+a_{pk}\frac{\partial w_{ik}}{\partial Y})+\frac{\partial b_{i}}{\partial Y}\} \qquad \text{(EQ 31)}$$

The *w's* are the weights on the connections between nodes and *b* is the bias term associated with a node; *k* runs over input, feedback and hidden units.

The activations of the input units are constant for each pattern so their derivatives with respect to anything are zero. The activations of the feedback units of pattern *p* are the activations of the hidden units of the previous pattern *p*-1. Likewise, the derivatives of activations of the feedback units of pattern *p* are the derivatives of the hidden units of the pattern *p*-1.

We still must consider the derivatives of the activations of the hidden units. The equation for these is the same as the previous one with *i* running over hidden units and *k* running over input and feedback units. The algorithmic bottleneck is computing derivatives for the gradient vector.

$$da_{pr} = 3\left[\frac{(t_{pr} - a_{pr})^2}{(1 - a_{pr}^2)} a_{pr} - (t_{pr} - a_{pr})\right] \qquad \text{(EQ 32)}$$

$$dh_{pr} = -1.5\,(1 - a_{pr}^2) \qquad \text{(EQ 33)}$$

Buffers for computing derivatives of the activations are *bh* and *ba* for the hidden and output layers, respectively. Multipliers used in computing derivatives are in *dh* and *da* and so,

$$bh_{pr} = dh_{pr}\,(\vec{w}_r \bullet \vec{q}_{pki} + \delta_{kr} a_{pi}) \qquad \text{(EQ 34)}$$

where the subscript *r* runs over units in the layers, and

$$ba_{pr} = da_{pr} \vec{w}_r \bullet \,(\vec{b}h_p \wedge \vec{q}_{pki}) \qquad \text{(EQ 35)}$$

where the δ is the Kronecker function and the vector *q* contains the derivatives of the feedback activations for pattern *p* with respect to the weight parameter. The subscripts *k* and *i* identify the weight parameter. The subscript *r* runs over the hidden layer. The weight vector *w* connects unit *r* to its predecessor units. The vector (*bh*^*q*) is a concatenation of the buffer *bh* and the vector *q*. Finally,

$$\frac{\partial \Phi}{\partial w_{ki}} = \sum_p \sum_r ba_{pr} \qquad \text{(EQ 36)}$$

where w is similarly defined, subscript k indexes the hidden layer, subscript i indexes the input and feedback layers and subscript r indexes the output layer.

The updated feedback derivatives are:

$$\vec{q}_{p+1,ki} = \vec{b}h_p \tag{EQ 37}$$

The q vectors are initialized to zero for the first pattern in a sequence. For k in the output layer and i in its predecessors

$$\frac{\partial \Phi}{\partial w_{ki}} = \sum_p da_{pk}a_{pi} \tag{EQ 38}$$

The bottleneck in doing a conjugate gradient iteration is evaluating the gradient of the error function and the bottleneck in computing the gradient involves the feedback derivatives. We believe we have created data structures for d, b, q and w vectors that place their values at a "unit stride" [GL89] apart so that our code may be nearly optimized for this critical section of derivative computation for a machine architecture that allows pipelined floating point computation. We hope to investigate further optimizations which take advantage of the capabilities of the various machine architectures that allow for concurrent floating point computation.

4.2 Feedback Derivatives for RAAMs

RAAMs are designed to use auto-associative training to evolve a collection of representations for structures. The networks are related to SRNs, but by definition are not allowed to have skip connections. Figure 3 provides an illustration of the architecture. Therefore, most of the results for RAAMs are identical with those of SRNs if this it taken into account. The major difference is in how one handles the output units which correspond to the distributed representation of the target. We isolate that part of the output to get the Kalman-Kwasny error function for it:

$$\Phi_{RAAM} = \sum_T \sum_k \frac{(RAAM_{Tk} - RAAM'_{Tk})^2}{1 - RAAM'^2_{Tk}} \qquad \text{(EQ 39)}$$

where T runs over the RAAMs in a sequence, k runs over the RAAM units, $RAAM_{Tk}$ is the target and input value and $RAAM'_{Tk}$ is the output value. Since $RAAM_{Tk}$ depends on the network parameters, we add a correction term to EQ 30 to get

$$\frac{\partial \Phi_{RAAM}}{\partial Y} = SRN' + \sum_T \sum_k \frac{2\,(RAAM_{Tk} - RAAM'_{Tk})}{(1 - RAAM'^2_{Tk})} \frac{\partial RAAM_{Tk}}{\partial Y} \qquad \text{(EQ 40)}$$

Where SRN' is the term which corresponds to EQ 30. We reported in [KKW93] that without this correction term, training for RAAMs usually does not converge and the error function often oscillates. A sample problem for which EQ 40 is important is a five level stack with a set of 3 symbols. Again without the correction term the RAAM version of TRAINREC oscillates badly but with it convergence is rapid.

FIGURE 3 ABOUT HERE

4.3 Settling and Voting

We used both settling and voting to aid in the training of a recurrent network that successfully identifies which of two languages is being spoken. The language identification problem uses raw speech signals and therefore is an example of a class of problems in which a possibly noisy sequence of inputs all predict the same output while the lengths of the sequences may vary.

Settling is a technique in which several patterns occurring initially in a sequence are not counted toward the error or its derivatives. Our reasoning is that the initial pattern in the recurrent sequence, chosen by us, is rather arbitrary and by not counting the contributions of the prefix the network is allowed to “settle” into the correct state. While settling does not incorporate the derivative terms of the prefix it stores them so that the pattern after the prefix takes account of the settling.

Voting is a technique of counting each prediction of the common output of a sequence as a vote on the common prediction. By using the binomial distribution we can use a relatively low frequency of correct prediction to bootstrap to a nearly perfect prediction in the case of a binary decision. If N is the number of patterns in a sequence (exclusive of the settling prefix) then CP is the cumulative probability that just more than half the N possible votes are for the correct prediction. The equation below gives a formula for computing CP, while Table 1 gives the minimum value of N for which $CP > 0.999$ for a given value of the probability of correct prediction p.

$$CP = \sum_{k = \lceil N/2 \rceil + 1}^{N} p^k (1-p)^{N-k} \binom{N}{k} \qquad \text{(EQ 41)}$$

No doubt that for the sequences mentioned, the independence of events assumed for the binomial distribution is violated. But for a sequence of events in which each event is supposed to predict the same output, we can think of the N column as being an upper bound of the length of sequence needed for virtually always correct prediction.

TABLE 3. Cumulative Binomial Probabilities

p	N	CP
0.6	246	0.99906
0.65	108	0.99904
0.7	60	0.99909
0.75	38	0.99922
0.8	24	0.99902
0.85	18	0.99949
0.9	12	0.99946

In [WKKE93] we reported on a recurrent network which was trained by TRAINREC to identify which of two languages a speaker was using. The overall performance on patterns in the test set was 78.9%. The worst case was 100/190 where 96/190 would have been acceptable. Use of voting on 190 patterns in a sequence allowed us to predict all of the test sequences correctly. This experiment strongly supports the contention that voting is valuable when working with such sequences.

4.4 Counting Unique Context States to Measure Generalization

Recurrent training sequences often come from mechanisms that tend to generate similar if not identical patterns as subsequences. During training, therefore, identical presentations can occur which distort the degree of emphasis placed on those presentations. Given a collection of pattern sequences, it can easily be determined with a little bookkeeping whether the pattern has occurred previously within the current sequence. Over an entire epoch, the number of unique states can be determined and serves as a useful upper bound on the number of feedback units required to distinguish the states. If S is the number of states occurring over an epoch of training patterns, then a useful heuristic for selecting the number of feedback units, F, is $F \leq \lceil lgS \rceil$. This comes from the observation that using extreme values only would permit the F units to uniquely represent this number of states as a binary value. Presumably the feedback units would be utilized in a more distributed manner, but selecting more feedback units than this would certainly over-specify the network leading to poor generalization.

In our NETTalk experiment, there were 5,523 patterns, but only 5,208 of those led to unique states. Therefore, no more than 13 feedback units should be required, which we have verified empirically. In training a recurrent neural network to be a deterministic parser for English [JKK93], we found that the 7,659 patterns resulted in 5,433 unique states which suggested an initial choice of 13 feedback units. This too has been verified experimentally. In the parser, in fact, a large number of context states in the test set do not appear in the training set, and yet 97% of the predictions on the states unique to the test set were correct. These results indicate the value of counting unique context states as a measure of generalization.

5.0 Summary and Conclusions

In this paper we have reviewed a number of discoveries, methods, and techniques that we have found valuable in our effort to develop various neural network models and apply them to real problems. The analysis of these findings is argued from basic principles and supported by numerous experiments using what we have discussed here to solve actual problems.

Each idea was distilled over several years of research from numerous ideas that were tried and failed or did not measure up as well as what we have presented. Each idea taken independently may not seem very profound, but taken collectively all of them fit together into a powerful set of training tools that have permitted the training and development of some large and complicated networks. Only through these techniques was it possible to train some of these networks using workstation-class machines and within a reasonable time frame.

Acknowledgments

Several people have contributed ideas and comments on this work and on this paper. William E. Ball and John W. Clark were especially helpful in comments on an earlier draft of this paper. Aurorita Abella, Ephrem Bartholomeos and Nancy Chang each served as Summer Undergraduate Research Assistants in this project under funding under NSF Grant No. CDA-9123643 and made valuable contributions toward this work. Ron Cytron graciously permitted us to run many of our training runs on his IBM RS 6000 machine without which very little of this work would have been accomplished.

References

[B92] E. Barnard, Optimization for training neural nets, *IEEE Trans. Neural Networks*, Vol. 3, No. 2, March 1992, pp. 232-241.

[E88] J. L. Elman, Finding Structure in Time, CRL Technical Report 8801, Center for Research in Language, University of California, San Diego, 1988.

[GGJ91] Shelly D. D. Goggin, Karl E. Gustafson and Kristina M. Johnson, An Asymptotic Singular Value Decomposition Analysis of Nonlinear Multilayer Neural Networks, Proceedings of the International Joint Conference on Neural Networks, July 1991, v. 1, pp. 785-790.

[GL83] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, pp 16-20 and 285-295, The Johns Hopkins University Press, 1983.

[GL89] Gene H. Golub and Charles F. Van Loan, *Matrix Computations 2d Edition*, page 42, The Johns Hopkins University Press, 1989.

[JKK93] Sahnny Johnson, Stan C. Kwasny and Barry L. Kalman, An Adaptive Neural Network Parser, forthcoming.

[K90] Barry L. Kalman, Superlinear Learning in Back-Propagation Neural Networks, Technical Report WUCS-90-21, Washington University, June, 1990.

[KH91] Gary M. Kuhn and Norman P. Herzberg, *Some Variations on Training of Recurrent Networks*, Academic Press, 1991.

[KK91] Barry L. Kalman and Stan C. Kwasny, A Superior Error Function for Training Neural Networks, Proceedings of the International Joint Conference on Neural Networks, July 1991, v. 2, pp. 49-52.

[KK92] Barry L. Kalman and Stan C. Kwasny, Why Tanh: Choosing a Sigmoidal Function, Proceedings of the International Joint Conference on Neural Networks, June 1992, v. IV, pp 578-581.

[KKA93] Barry L. Kalman, Stan C. Kwasny and Aurorita Abella, Decomposing Input Patterns to Facilitate Training, Proceedings of the World Congress on Neural Networks, Portland, OR, July 1993, forthcoming.

[KKEW92] Stan C. Kwasny, Barry L. Kalman, A. Maynard Engebretson, and Weilan Wu, Identifying Language from Speech: An Example of High-Level, Statistically-Based Feature Extraction, Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, July, 1992, pp. 909-914.

[KKC93] Stan C. Kwasny, Barry L. Kalman and Nancy Chang, Distributed Patterns as Hierarchical Structures, Proceedings of the World Congress on Neural Networks, Portland, OR, July 1993, forthcoming.

[LH92] Samuel E. Lee and Bradley R. Holt, Regression Analysis of Spectroscopic Process Data Using a Combined Architecture of Linear and Nonlinear Artificial Neural Networks, Proceedings of the International Joint Conference on Neural Networks, June 1992, v. IV, pp 549-554.

[M80] Mitchell Marcus, *A Theory of Syntactic Recognition for Natural Language,* MIT Press, 1980.

[MGB74] Alexander M. Mood, Franklin A. Graybill and Duane C. Boes, *Introduction to the Theory of Statistics*, 3rd ed., McGraw-Hill, 1974.

[P71] E. Polak, *Computational Methods in Optimization: A Unified Approach*, Academic Press, 1971.

[P90] Jordan Pollack, Recursive Distributed Representations, *Artificial Intelligence 46*, 77-105.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetter-ing, *Numerical Recipes in C*, Cambridge University Press, 1988.

[SR87] Terrence J. Sejnowski and Charles R. Rosenberg, Parallel Networks that Learn to Pronounce English Text, *Complex Systems 1*, 1987, pp 145-168.

[SCM88] David Servan-Schreiber, Axel Cleeremans and James L. McClelland, Encoding Sequential Structure in Simple Recurrent Networks, Technical Report CMU-CS-88-183, Carnegie Mellon University, November, 1988.

[VB93] Jacques de Villiers and Etienne Barnard, Backpropagation Neural Nets with One and Two Hidden Layers, IEEE Transactions on Neural Networks, vol 4., No. 1, January 1993, pp 136-141.

[WB90] Darrell Whitley and Christopher Bogart, The Evolution of Connectivity: Pruning Neural Networks Using Genetic Algorithms, Proceedings of the International Joint Conference on Neural Networks, January 1990, v. 1, pp 134-7.

[WK90] Sholom M. Weiss and Casimir A. Kulikowski, *Computer Systems That Learn*, Morgan Kaufmann Publishers, 1990.

[WKKE93] Weilan Wu, Stan C. Kwasny, Barry L. Kalman and A. Maynard Engebretson, Identifying Language from Raw Speech: An Application of Recurrent Neural Networks, Proceedings of the Midwest Artificial Intelligence and Cognitive Science Conference, April 1993, pp 53-57.

[XHT90] Qiuzhen Xue, Yuhen Hu and Willis J. Tompkins, Analyses of Hidden Units of Back Propagation Model by Singular Value Decomposition, Proceedings of the International Joint Conference on Neural Networks, January 1990, v. 1, pp 739-742.

FIGURE 1. Feedforward Network with Skip Connections

Units are represented by circles, layers are represented by rounded boxes, and weights are represented by arrows. One or more arrows connecting two layers means the layers are completely connected.

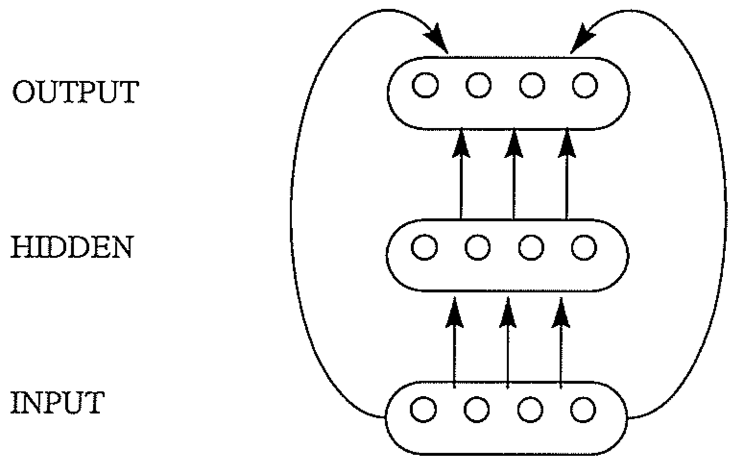


FIGURE 2. Simple Recurrent Network

The thick arrow represents feedback. Activation patterns are copied back on successive steps.

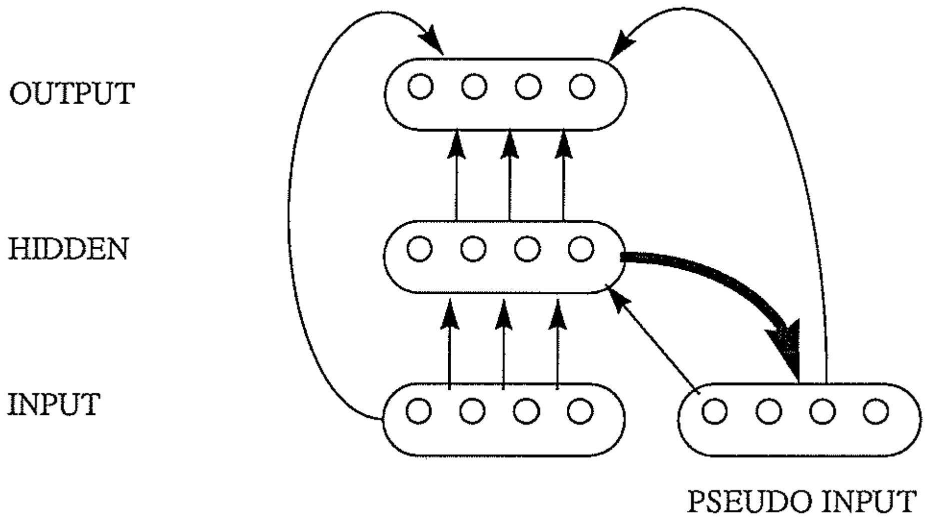
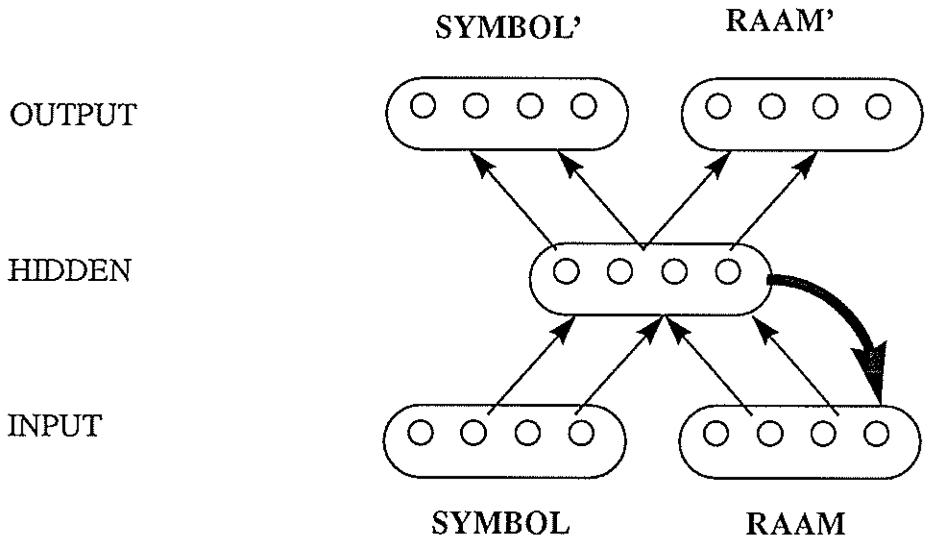


FIGURE 3. Sequential Recursive Auto-Associative Network

Auto-associativity assures that the input and target patterns are identical. This figure assumes that on each iteration a single symbol is encoded and presented to the network along with the hidden-layer (RAAM) activation pattern of the previous iteration.


SYMBOL'
RAAM'
OUTPUT
HIDDEN
INPUT
SYMBOL
RAAM