

High Performance Training of Feedforward & Simple Recurrent Networks

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-squares-error function. We argue for skip (shortcut) connections where appropriate and the preference for a bipolar 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 continues to increase, particularly as researchers have realized the importance of recurrent networks. Recurrent architectures have the advantage of being able to remember, to 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. For example Servan-Schreiber et. al.[26] discuss usage of recurrent networks as representations of state machines. 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, simple recurrent neural networks and recursive auto associative memories[21]. The emphasis in TRAINREC is on training networks with recurrent architecture 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[17]) 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 our motivation.

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 choosing an error function to determining the input feature space. Over the past several years, we have collected evidence which documents the effects these discoveries and techniques as TRAINREC has improved. We present this evidence along with theoretical justification and practical implications.

In this subsection we present some of the assumptions, techniques and features that are usually accepted 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 several aspects of them. Below we present a list of standard features. In later sections we discuss more details for improving these features.

- **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. See Section 2.1.
- **Layered connectivity.** Most networks use layers of units with each layer completely connected to the preceding layer as shown in Figure 1. 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 and Section 2.2.

FIGURE 1 ABOUT HERE

- **The [0, 1] interval.** The bipolar([-1, 1]) interval is better for Boolean functions and many other problems. While a simple affine transformation can map between the two intervals, avoiding zero as an extreme value can have an important effect on training. Section 2.3 contains more discussion.
- **Intense feature analysis.** Much time may be wasted in deciding the necessary and sufficient set of input features and their values. We show how a set of sufficient features can be analyzed and reduced through singular value decomposition. See Section 2.3.
- **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. See Section 2.4.
- **Mean square error stopping criterion.** The network can be very wrong on a few important unit activations even though the mean square error is very small. This can lead to poor training especially for categorization problems. See Section 2.4.
- **Pattern based training.** From our experience, particularly with 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. See Section 3.1.
- **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. See Section 3.1.
- **Overly constrained architectures for simple recurrent neural network.** Because of backprop, most simple recurrent neural networks 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 often desired as discussed in Section 4.1.
- **Static targets for Recursive Auto-Associative Memories.** Recursive auto associative memories cleverly use auto-associativity to permit representation of complex structures as distributed patterns (see [21] and [14]). In our experience, the effect of the variability of the target is usually not properly considered under standard implementations of backprop. See Section 4.2.

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 self-scaling error function has been a key element in the success of this effort. See Section 2.1.
- **Use of singular value decomposition to preprocess input.** We have found it productive to use singular value decomposition[5] 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. See Section 2.3.
- **The bottleneck of recurrent network training.** By analyzing the algorithm, we found that evaluating derivatives is a bottleneck for training recurrent networks. By reducing the frequency of derivative computations, efficiency increases greatly. See Section 3.1.
- **Use of network voting.** Separately trained networks can be combined by voting schemes to improve performance. See Section 3.4.
- **Use of settling.** In training a simple recurrent neural network for tasks involving noisy data we found that allowing the network to “settle” over a prefix of patterns is very helpful. See Section 4.3
- **Use of sequential voting.** We found that a voting scheme based on the cumulative binomial distribution may bootstrap a weakly performing network into an asymptotically perfect separator. See Section 4.3.

1.3 Overview

The remainder of the article contains four sections. In Section 2 we discuss the methods which improve efficiency in a static sense without affecting the parameters of the network.

In Section 3 we describe the methods which increase efficiency dynamically during training. In Section 4 we present special techniques we have used for specific network architectures and specific problems.

2.0 Statics

In this section we discuss techniques which do not depend directly upon the network parameters, such as size, feedback, etc.

2.1 The Self-scaling Error Function.

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[10] 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:

$$\Phi = \sum_p \sum_k g_{pk} \tag{EQ 1}$$

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:

$$\lim_{a \rightarrow t} g = 0 \tag{EQ 2}$$

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

$$\lim_{a \rightarrow \tilde{t}} g = \infty \quad (\text{EQ 3})$$

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

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

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

$$\lim_{a \rightarrow \tilde{t}} |g'| = \infty \quad (\text{EQ 5})$$

If g_{pk} is chosen as the square of the error, e_{pk}^2 , where $e_{pk} = (t_{pk} - a_{pk})$, then Φ is the sum of squares error function. 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. If the property expressed by EQ 3 is not satisfied then situations occur in which the gradient of Φ is zero when $\Phi > 0$. This happens when one or more of the outputs is exactly opposite to its desired target value. This situation is an exterior saddle point. Training situations occur where one has two output classes and one occurs much less frequently than the other but it is desirable to be very often correct in predicting members of the less frequent class. For the sum of squares error function the conjugate gradient algorithm often terminates with a small value of Φ but at an exterior saddle point for a pattern in the less frequent class. Under the given constraints this is an attractively dangerous termination of training. An error function which satisfies EQ 3 can not get into the described difficulty. If the property expressed by EQ 5 is satisfied then approaching one of the exterior saddle points of the sum of squares error function is similar to the way a child learns by touching a hot stove - training moves away from such a region quickly.

In order to satisfy all four properties, we have been investigating a family of scaled error functions of the form $g_{pk} = \frac{e^2}{s_{pk}}$ under the conditions above. We have added a scaling factor s_{pk} , which depends on pattern and output unit. We now endeavor to determine a suitable scaling factor which must also depend 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:

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

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 the sigmoidal:

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

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:

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

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

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

This relation leads to the scaling function:

$$s_{pk} = 1 - a_{pk}^2 \quad (\text{EQ 10})$$

We have shown in [11] that selecting the value:

$$\lambda = 1.5 \tag{EQ 11}$$

maintains an equitable scaling of weight layers for training purposes. Using an argument parallel to that of Rigler et al.[24] that arose from the observation that if the expected value of the derivative of the sigmoidal $E[\sigma']$ is 1 over the weights on connections between layers of a network then training does not attenuate on layers far from or close to the output layer. For layer n (by counting back from the output layer) and from EQ 8 we get

$$E \left[\lambda^n \prod_{k=0}^{n-1} \left(1 - \sigma^2(x_{pn-k}) \right) \right] = 1 \tag{EQ 12}$$

By integrating over the interval [-1, 1] with respect to σ we get $\lambda = 1.5$.

In [10] we reported on the performance of our error function on the problem of training a feedforward network to be a deterministic parser (see Marcus [16, 272-276]) 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 [10]. These results are strong evidence for using the self-scaling error function.

TABLE 1. Results for Medium Grammar Problem

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

The entries of ∞ in Table 1 mean that the experiment always terminated at an exterior saddle point with incorrect predictions. This did not occur for the other experiments.

2.2 Using Skip Connections

Most problems have a significant linear component in their solutions. Some require only a 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 [15].

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 [29] 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 [28]. If one is testing for the optimal number of hidden units for a network then the case of zero hidden units is a more natural limiting case than for a network without skip connections. Of course, TRAINREC permits disconnection of inputs and outputs by user choice. This is an essential choice, of course, for recursive auto associative memory networks.

In [12] we reported on several problems for 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 ala Elman[2]. More general recurrent networks are discussed by Williams and Zipser[32]. The nodes to which the outputs of the hidden units are fed back we call 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 [8] 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 previously reported[12] the use of singular value decomposition to preprocess input. We discussed the use of singular value decomposition along with an affine transformation to place the inputs in the interval $[-1, 1]$. We observe that in the case of complete Boolean functions that with the $[-1,1]$ interval the input columns are orthogonal while with the $[0,1]$ interval they are not, and that training with $[-1,1]$ is much faster than with $[0,1]$. (Note that complete here means all input patterns are used.) Use of singular value decomposition along with the $[-1,1]$ interval for inputs often allowed training to proceed when it was previously impossible. Singular value decomposition may also reduce the number of input units needed to represent the input without a significant effort in feature extraction. (Fausett [3, 309-312] also has reported uses for the $[-1,1]$ interval.) 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. Details of singular value decomposition are presented in [5, 427-435]. Uses of singular value decomposition to analyze the hidden layer are presented in [4] and [33].

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 singular value decomposition of A is:

$$A = UGV^T \quad (\text{EQ 13})$$

where U and V are orthonormal and G is diagonal. If singular value decomposition 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:

$$r_i = \min_{1 \leq k \leq p} Q_{ki} \quad (\text{EQ 14})$$

$$s_i = \max_{1 \leq k \leq p} Q_{ki} \quad (\text{EQ 15})$$

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

$$d_i = -\frac{c_i}{2} (s_i + r_i) \quad (\text{EQ 17})$$

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

$$A'_{ki} = Q_{ki}c_i + d_i \quad (\text{EQ 18})$$

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 [12] 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:

$$w_{ij} = \sum_k Z_{ik} w'_{kj} \quad (\text{EQ 19})$$

$$b_j = \sum_k w'_{kj} d_k + b'_j \quad (\text{EQ 20})$$

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 singular value decomposition-affine preprocessing. If one uses the orthonormal properties of V with EQ 19 and EQ 20, it is easy to derive transformations to compute w' and b' from the singular value decomposition-affine preprocessing of the enlarged input set as follows:

$$w'_{kj} = \sum_i w_{ij} (V_{ik} / c_k) \quad (\text{EQ 21})$$

$$b'_j = b_j - \sum_k \sum_i (w_{ij} d_k V_{ik}) / c_k \quad (\text{EQ 22})$$

The value of this technique ranges from having no effect to being absolutely essential. 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, singular value decomposition 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. Singular value decomposition 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, singular value decomposition 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, singular value decomposition reduces the number of inputs from 38 to 34. Without the singular value decomposition-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 and best vector match. Each has its own usefulness.

With perfect match 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.

Vector best match 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. Vector best match works especially well when one output unit is designated as a default catchall for cases where the predicted category is none of those which come from the model. This is because every vector is orthogonal to the zero vector. Vector best match is used quite effectively, for example, in NetTalk [27].

In our work on recursive auto associative memories, 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 vector best match. The choice of the empty symbol is very important in this work. If we represent it by a vector of all -1's, our vector best match 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 vector best match very often is a better measure of learning than either perfect match 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 and Brent's derivative free line search. Since evaluation of derivatives for simple recurrent neural networks and recursive auto associative memories is very expensive, derivative free line search with the conjugate gradient algorithm provides a very efficient approach to training these networks. We also discuss an adaptive step size control technique which further enhances derivative free line search.

3.1 Conjugate Gradient Algorithm

We use the conjugate gradient algorithm[7], [20], [22], [23, 420-425]because it is quadratically convergent, it behaves as well as backprop in tough regions and it is possible to use less expensive derivative free line search for most of the training. Others (see, for example [1]) have independently found it very useful. Because of quadratic convergence it uses fewer epochs (often many fewer) than backprop especially if a Powell update[22] is used. The conjugate gradient algorithm requires linear storage (in weights) while Newton-style quadratically convergent methods require quadratic storage. The conjugate gradient algorithm’s generalization properties are very close to those of backprop. The results in Table 1 give strong evidence for the use of conjugate gradient algorithm over ordinary backprop.

3.2 Derivative-Free Line Search

The conjugate gradient algorithm requires a line search algorithm to find a local minimum in a particular direction for each iteration. Two line search algorithms are Brent[23, 404-405] and Dbrent [23, 406-408]. 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 derivative free line search. 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 conjugate gradient algorithm iteration which uses derivative and derivative free line search, respectively. The leading terms in the complexity expressions are by algorithmic analysis of flops:

$$Fe = 2p (H \times I + O \times (H + I)) \quad (\text{EQ 23})$$

$$De = 4p(H \times I + O \times (H + I)) \quad (\text{EQ 24})$$

With the Dbrent algorithm for linesearch we observe that one conjugate gradient algorithm 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,

$$\frac{Td}{Tf} = \frac{16}{11} \quad (\text{EQ 25})$$

One should note that every epoch in ordinary backprop is a derivative epoch. The ability of conjugate gradient algorithm to use derivative free epochs is alone enough to recommend it. With simple recurrent neural networks and recursive auto associative memories the improvement due to derivative free line search is much more noticeable.

Here we analyze simple recurrent neural networks where the number of feedback units is the same as the number of hidden units. The simple recurrent neural network result for Fe is:

$$Fe = 2p(H \times (I + H) + O \times (2 \times H + I)) \quad (\text{EQ 26})$$

In Section 4 we present the algorithm for computing derivatives which depend on feedback for a simple recurrent neural network. An analysis of that algorithm leads to:

$$De = Fe + 2pH^2((I + H) \times (H + 2 \times O)) \quad (\text{EQ 27})$$

With simple recurrent neural networks 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 conjugate gradient algorithm 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.

We now analyze recursive auto associative memories where $O = H + I$. The simple recurrent neural network version of recursive auto associative memories that we use is described in [14]. By definition, we are not able to use skip connections with our recursive auto associative memories. We get:

$$Fe = 4pH(I + H) \tag{EQ 28}$$

$$De = Fe + 2pH^2(H + I)(2H + I) \tag{EQ 29}$$

which leads to $\frac{Td}{Tf} = 8$, the same as for simple recurrent neural networks.

3.3 Adaptive Step Size Control

Line search requires a region known to contain a local minimum before it can start. The algorithm `mnbrak` [23, 400-401] locates such a region. It requires three points along the search direction to initialize it. At the beginning of conjugate gradient algorithm we use the ratio of the error function to the magnitude of the gradient as a start for `mnbrak`.

During the conjugate gradient algorithm we use adaptive step size control 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 conjugate gradient algorithm iteration to the next. If the actual step along the search direction is above 0.8 of the original interval length adaptive step size control doubles the size of the initial search range; if the actual step is below 0.2 of the original length, adaptive step size control halves the size of the initial search range. We observe that less than

50% of the derivative free line search epochs are needed when compared with that required by a fixed initial search range.

3.4 Tests of Generalization

Weiss and Kulikowski[30, 31-33] suggest that the technique of cross validation be used when only a modest amount of data is available, which is the case in most neural network investigations. Cross validation involves randomly dividing the data into several roughly equally sized sets. Each set is used as a testing set in turn while the remainder of the data is used for training. Overall performance is estimated by averaging over the performance of the testing sets. When copious data is available, only one testing set of about 20% of the data is needed. To reduce bias introduced by terminating training based on performance of training or testing sets we usually select a random hold out set which is in no way involved in training. The networks produced by cross validation are applied to the holdout data and majority voting is used to estimate “true” performance.

We test generalization by evaluating the network with the test set every k conjugate gradient algorithm 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. But predicting occurrences of the less frequent categories is often important and 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 [23, 628-632]) 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 [18, 452-461]. Construction of the confusion matrix is discussed in Section 2.4.

The third measure concerns maximizing performance of the most difficult case. We save the network whenever the observed lowest fraction correct in the test set categories increases as long as the performance on the training set is minimally acceptable. 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.

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.

3.5 Adjusting the Targets to Avoid Local Minima

We have found that adjusting the targets for the output units in the fashion of Gorse, Shepherd and Taylor[6] avoids local minima that stop the conjugate gradient algorithm if we hold the targets at ± 1 . We have modified the method slightly. We start with targets of ± 0.2 which avoids many of the local minima found with ± 1 and also avoids extremum found with all outputs = 0. If a local extremum is found we iteratively shrink the targets

down to ± 0.025 . This is found to be sufficient to obtain what appear to be global extrema. We then iteratively increase the targets to ± 1 . We have found that occasionally the solution gets better and never gets worse during the target increase. This phenomena was not reported in [6]. We attribute it to our choice of error function. Since this error function approaches infinity at the wrong end of the interval big errors encountered as the targets are increased may be dramatically reduced, which yields the observed improvements. This cannot happen with the ordinary squared error function.

4.0 Architecture 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 simple recurrent neural networks and recursive auto associative memories as well as how to compute them efficiently. We discuss the use of settling and voting to bootstrap the performance of a network we are using for language identification. In analyzing input pattern sequences for recurrent networks, we observed that portions of the sequences may be duplicated resulting in identical training subsequences. Once observed this problem is easily remedied.

4.1 Feedback Derivatives for Simple Recurrent Neural Networks

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 the algorithm and discuss how performance might be improved through the use of parallel architectures.

First, we show the derivatives for a simple recurrent neural network. Recall that the error function is given by EQ 1. For classification problems we use g_{pk} as defined by the scaling factor of EQ 10 and for infinite range linear output problems such as function fitting we use $g_{pk} = e^2_{pk}$ where $e_{pk} = (t_{pk} - a_{pk})$, p ranging over patterns and k over out-

put units. Target, t_{pk} , and activation, a_{pk} , are specific to an input pattern presentation and an output unit. The derivative equations are given by the generalized delta rule[25] with an additional term due to the feedback units. The delta terms for the output units are given by:

$$\Delta_{pk} = \begin{cases} (-2e_{pk}) & \text{Linear Output} \\ 3(g_{pk}a_{pk} - e_{pk}) & \text{Classification} \end{cases} \quad (\text{EQ 30})$$

For the hidden units the delta terms are given by:

$$\Delta_{pj} = 1.5 \left(1 - a_{pj}^2 \right) \sum_k \Delta_{pk} w_{kj} \quad (\text{EQ 31})$$

where j ranges over the hidden units and k is as in EQ 30 and w_{kj} is the weight of the connection between units k and j . For the feedback units the delta terms are given by:

$$\Delta_{pj} = \sum_k \Delta_{pk} w_{kj} \quad (\text{EQ 32})$$

where j ranges over the feedback units and k ranges over the hidden and output units.

The only derivative equation that affects the gradient and which involves feedback units is:

$$\frac{\partial \Phi}{\partial w_{ji}} = \sum_p \left[\Delta_{pj} a_{pi} + \sum_l \Delta_{pl} \frac{\partial a_{pl}}{\partial w_{ji}} \right] \quad (\text{EQ 33})$$

where j ranges over the hidden units and i ranges over the input and feedback units and l ranges over the feedback units. The remainder of the terms in the gradient are given by the generalized delta rule:

$$\frac{\partial \Phi}{\partial w_{ki}} = \sum_p \Delta_{pk} a_{pi} \quad (\text{EQ 34})$$

Where k ranges over output units and i ranges over input, feedback and hidden units.

Equations for the bias derivatives are obtained by replacing the appropriate a_{pi} by 1.

Because of the recurrence in the network we still must consider the derivatives of the activations of the hidden units which become the derivatives of the activations of the feedback units for the next pattern in a sequence. The equation for these is:

$$\frac{\partial a_{p+1,l}}{\partial w_{ji}} = \delta_{li} a_{pi} + \sum_s \frac{\partial a_{ps}}{\partial w_{ji}} w_{js} \quad (\text{EQ 35})$$

where j ranges over the hidden units, i , l and s range over the feedback units. δ_{li} is the Kronecker delta function which is one if $i = l$ and zero otherwise. (N.B., the new feedback derivatives should be preserved until all feedbacks associated with a particular weight have been computed.)

The bottleneck in doing a conjugate gradient iteration is evaluating the gradient of the error function which, in turn, involves feedback derivatives. Assuming H , I and O defined as in EQ 27 and the number of feedback units is H , EQ 35 explains why the feedback part of the gradient for a recurrent network is expensive to evaluate as compared to the gradient computation of a non recurrent network. There are H^3 terms of the type on the left side of the equation and each one involves the sum over H terms. If the sum part of EQ35 is evaluated as a scalar product the code to implement EQ 35 can be optimized for a scalar pipelined processor[5, 36-37]. Even more of the potential concurrency in EQ 35 can be implemented on a shared memory multi-processor computer as demonstrated in McCann and Kalman[19].

4.2 Feedback Derivatives for Recursive Auto Associative Memories

Recursive auto associative memories are designed to use auto-associative training to evolve a collection of representations for structures. The networks are related to simple recurrent neural networks, but by definition preclude the use of skip connections. Figure 3 provides an illustration of the architecture. Therefore, most of the results for recursive auto associative memories are identical with those of simple recurrent neural networks if this is 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 self-scaling error function for it:

$$\Phi_{RAAM} = \sum_T \sum_k \frac{(RAAM_{Tk} - RAAM'_{Tk})^2}{1 - RAAM_{Tk}^2} \quad (\text{EQ 36})$$

where T ranges over the recursive auto associative memories in a sequence, k ranges over the recursive auto associative memory 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 34 to get

(EQ 37)

$$\frac{\partial \Phi_{RAAM}}{\partial w_{ki}} = SRN' + \sum_T \sum_k \frac{2(RAAM_{Tk} - RAAM'_{Tk})}{(1 - RAAM_{Tk}^2)} \frac{\partial RAAM_{Tk}}{\partial w_{ki}}$$

Where SRN' is the derivative of the term which corresponds to EQ 34. We reported in [14] that without this correction term, training for recursive auto associative memories usually does not converge and the error function often oscillates. A sample problem for which EQ 37 is important is a five level stack with a set of 3 symbols. Again without the correction

term the recursive auto associative memory 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[13]. 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 a better initial 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 .

$$CP = \sum_{k = \lceil N/2 \rceil + 1}^N p^k (1-p)^{N-k} \binom{N}{k} \quad (\text{EQ 38})$$

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 [31] 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 $F \leq \lceil \lg S \rceil$. 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[27] 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 [9], 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

- [1] E. Barnard(1992), Optimization for Training Neural Nets, *IEEE Trans. Neural Networks*, **3**(2), 232-241.

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

- [3] Laurene Fausett(1994), *Fundamentals of Neural Networks: Architectures, Algorithms and Applications*, Prentice Hall.

- [4] Shelly D. D. Goggin, Karl E. Gustafson and Kristina M. Johnson, An Asymptotic Singular Value Decomposition Analysis of Nonlinear Multilayer Neural Networks(1991), *Proceedings of the International Joint Conference on Neural Networks*, **1**, 785-790.
- [5] Gene H. Golub and Charles F. Van Loan(1989), *Matrix Computations 2d Edition*, The Johns Hopkins University Press.
- [6] D. Gorse, A. Shepherd and J. G. Taylor(1994), A Classical Algorithm for Avoiding Local Minima, *Proceedings of the World Congress on Neural Networks*, San Diego, CA, **III**, 364-369.
- [7] Barry L. Kalman(1990), *Superlinear Learning in Back-Propagation Neural Networks*, Technical Report **WUCS-90-21**, Washington University.
- [8] Gary M. Kuhn and Norman P. Herzberg(1991), *Some Variations on Training of Recurrent Networks*, Academic Press.
- [9] Stan C. Kwasny, Sahnny Johnson and Barry L. Kalman(1994), Recurrent Natural Language Parsing, *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, Atlanta, GA, **1**,525-530.
- [10] Barry L. Kalman and Stan C. Kwasny(1991), A Superior Error Function for Training Neural Networks, *Proceedings of the International Joint Conference on Neural Networks*, **2**, 49-52.
- [11] Barry L. Kalman and Stan C. Kwasny(1992), Why Tanh: Choosing a Sigmoidal Function, *Proceedings of the International Joint Conference on Neural Networks*, **IV**, 578-581.

- [12] Barry L. Kalman, Stan C. Kwasny and Aurorita Abella(1993), Decomposing Input Patterns to Facilitate Training, *Proceedings of the World Congress on Neural Networks*, Portland, OR, **III**, 503-506.
- [13] Stan C. Kwasny, Barry L. Kalman, A. Maynard Engebretson, and Weilan Wu, Real-time Identification of Language from Raw Speech Waveforms(1993), *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*, Lawrence Erlbaum Associates, 161-167.
- [14] Stan C. Kwasny, Barry L. Kalman, Tail-recursive Distributed Representations and Simple Recurrent Networks, *Connection Science*, **7**(1), 1995, pp. 61-80.
- [15] Samuel E. Lee and Bradley R. Holt(1992), 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*, **IV**, 549-554.
- [16] Mitchell Marcus(1980), *A Theory of Syntactic Recognition for Natural Language*, MIT Press.
- [17] J. L. McClelland and D. E. Rumelhart(1988), *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs and Exercises*, MIT Press.
- [18] Alexander M. Mood, Franklin A. Graybill and Duane C. Boes(1974), *Introduction to the Theory of Statistics*, 3rd ed., McGraw-Hill.
- [19] Peter J. McCann and Barry L. Kalman(1994), Parallel Training of Simple Recurrent Neural Networks, *Proceedings of the IEEE World Congress on Computational Intelligence(WCCI '94)*, Orlando, FL, **I**, 167-170.

- [20] E. Polak(1971), *Computational Methods in Optimization: A Unified Approach*, Academic Press.
- [21] Jordan Pollack, Recursive Distributed Representations(1990), *Artificial Intelligence*, **46**, 77-105.
- [22] M. J. D. Powell(1977), Restart Procedures for the Conjugate Gradient Method, *Mathematical Programming*, **12**(2), 241-254.
- [23] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling(1988), *Numerical Recipes in C*, Cambridge University Press.
- [24] A. K. Rigler, J. M. Irvine and T.P. Vogl(1991), Rescaling of Variables in Back Propagation Learning, *Neural Networks*, **4**, 225-229.
- [25] David E. Rumelhart, James L. McClelland and the PDP Research Group(1986), *Parallel Distributed Processing*, **1**, The MIT Press.
- [26] David Servan-Schreiber, Axel Cleeremans and James L. McClelland(1988), *Encoding Sequential Structure in Simple Recurrent Networks*, Technical Report **CMU-CS-88-183**, Carnegie Mellon University.
- [27] Terrence J. Sejnowski and Charles R. Rosenberg(1987), Parallel Networks that Learn to Pronounce English Text, *Complex Systems*, **1**, 145-168.
- [28] Jacques de Villiers and Etienne Barnard(1993), Backpropagation Neural Nets with One and Two Hidden Layers, *IEEE Transactions on Neural Networks*, **4**(1), 136-141.
- [29] Darrell Whitley and Christopher Bogart(1990), The Evolution of Connectivity: Pruning Neural Networks Using Genetic Algorithms, *Proceedings of the International Joint Conference on Neural Networks*, **1**, 134-137.

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

[31] Weilan Wu, Stan C. Kwasny, Barry L. Kalman and A. Maynard Engbretson(1993), Identifying Language from Raw Speech: An Application of Recurrent Neural Networks, *Proceedings of the Midwest Artificial Intelligence and Cognitive Science Conference*, **1**, 53-57.

[32] Ronald. J. Williams and David Zipser(1989), A Learning Algorithm for Continually Running Fully Recurrent Neural Networks, *Neural Computation*, **1**, 270-280.

[33] Qiuzhen Xue, Yuheng Hu and Willis J. Tompkins(1990), Analyses of Hidden Units of Back Propagation Model by Singular Value Decomposition, *Proceedings of the International Joint Conference on Neural Networks*, **1**, 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.

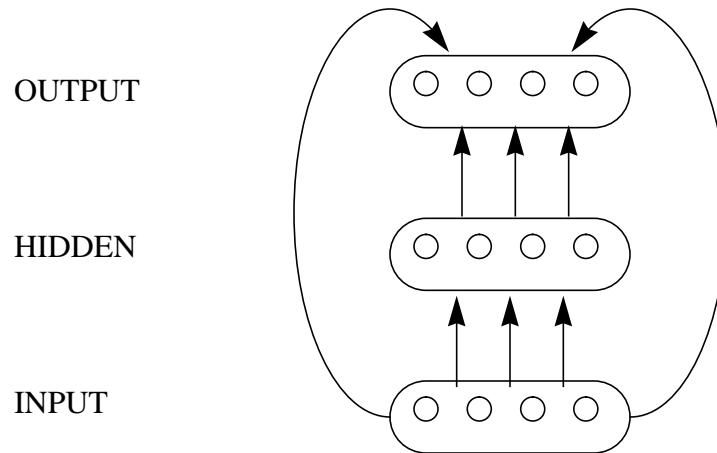


FIGURE 2. Simple Recurrent Network

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

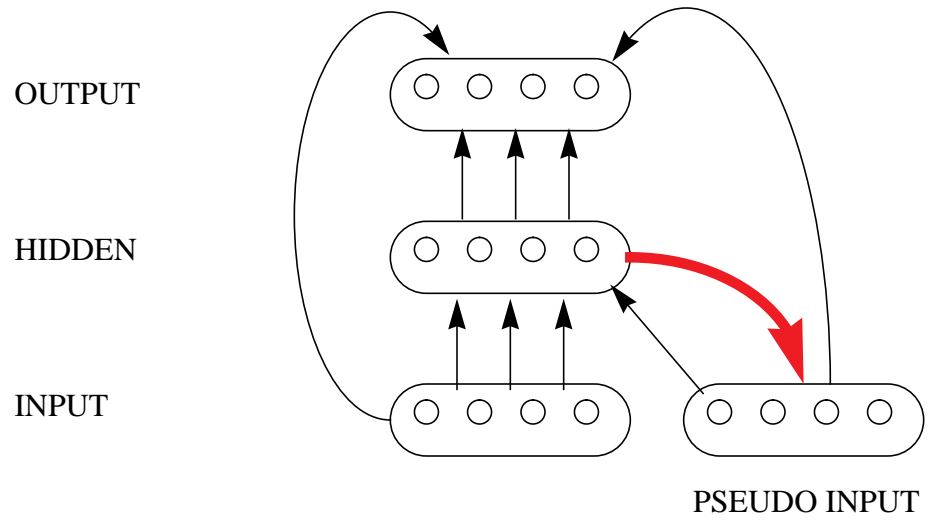
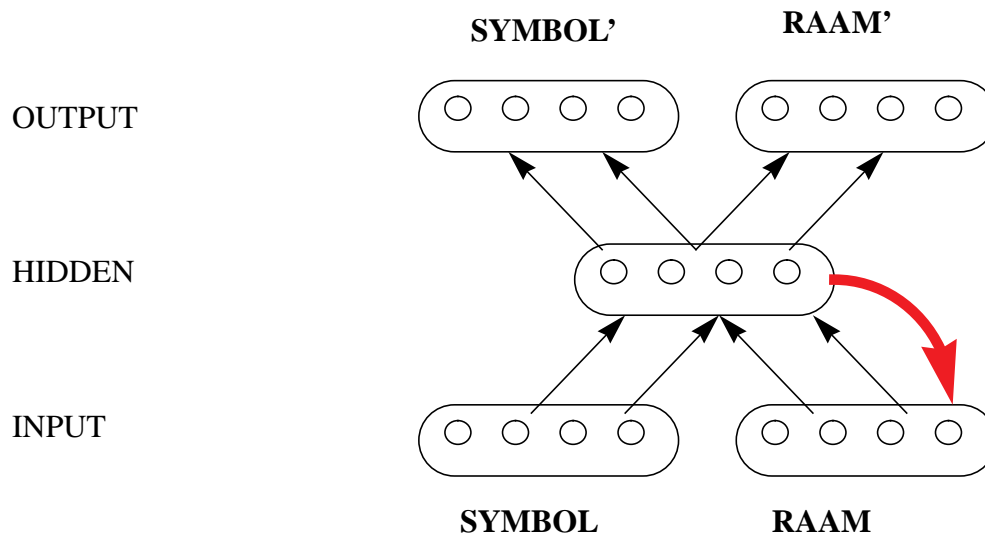


FIGURE 3. Sequential Recursive Auto-Associative Network

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 (recursive auto associative memory) activation pattern of the previous iteration.



**High Performance Training of
Feedforward & Simple Recurrent Networks**

Barry L. Kalman and Stan C. Kwasny

WUCS-94-29

October 1994

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