

Recurrent Neural Networks for Sequence Production

Joachim Diederich* Markus Wasserschaft**

*Neurocomputing Research Centre
School of Computing Science
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434
Brisbane Q 4001 Australia

**German National Research Center for Computer Science (GMD)
Institute for Applied Information Processing (FIT)
P.O. Box 1316
D-5205 St. Augustin 1 Germany

Abstract

In an empirical study, simple recurrent networks (SRNs, Elman, 1990) and recurrent cascade correlation networks (RCC, Fahlman, 1991) are compared with respect to a sequence production task. The application system that includes these networks extracts relevant information from incoming e-mail messages during a training period, learns the reaction to the incoming message, i.e., a sequence of user actions, and uses the learned representation for the proposal and preparation of user reactions. The performance of both network architectures in terms of network size and learning speed for a given data set is examined. Our results show that (1) RCC generates smaller networks with better performance compared to SRNs and (2) learns significantly faster than SRNs.

1 Introduction

In this paper, a comparison of two recurrent network architectures is presented. The performance of simple recurrent networks (SRN, Elman 1990) is compared to the recurrent cascade correlation architecture (RCC, Fahlman 1991). For both architectures, this study focuses on the required network size (number of hidden units) and the speed of the learning process to perform a given task.

The background is a neural network application (called NECAS), an advanced mail tool which runs on top of UNIX mail. Here, relevant information from incoming messages is extracted and neural networks learn user actions in response to these messages. In Diederich et al. (1992) it is shown that SRNs are capable of learning this task. A problem with SRNs is that the size of the network (number of hidden units and layers) has to be chosen in advance, which may lead to inappropriate choices (e.g., too many hidden units) and consequently to bad generalization.

RCC, however, incorporates an incremental learning

algorithm and is supposed to build reasonable small networks. The size of the generated RCC-network gives an indication of a reasonable size for a SRN.

In this comparison, a sequence production process is examined. More precisely: The relation between a pattern's position inside a sequence and the difficulty (i.e. number of pattern presentations) to learn it, is investigated.

Our objectives are the following:

1. Optimization of the size of the SRN using the results of the RCC experiments
2. Comparison of the performance of RCC vs. SRN with respect to our applications data set
3. Study the performance of the networks doing sequence production. For instance, we are interested in the following question: Given a sequence of training patterns and a recurrent network: Is there a relation between a patterns position inside a sequence and the difficulty (i.e., number of pattern presentations) to learn it?

In this paper, we first give an introduction to SRNs. Next, the particular networks we are using (called NECAS Recurrent Network; NECAS RN) plus the RCC-architecture is described. Section 3 provides an overview of the data set used for experimental purposes. Next, in section 4, the methodology used for evaluation is outlined and after the presentation and discussion of the results in section 5, we conclude with some remarks on future work.

2 The neural network architectures

2.1 Simple recurrent networks

A simple recurrent network for sequence production can emulate a finite-state machine and implements an output function f and a next-state function g . At each

moment in time, an action is chosen based on the current state s , and the state is then updated to allow the next action to be chosen (Jordan, 1990, p.115). The function f determines the output action x_n at time n ,

$$x_n = f(s_n, p)$$

and a function g determines the state s_{n+1} ,

$$s_{n+1} = g(s_n, p)$$

where both functions depend on the constant plan vector p as well as on the current state vector s_n . In a Jordan-Network (Jordan 1986, 1990), "plans," "states" and "actions" are distributed patterns of activation over separate groups of units. All feedforward connections implement f and all recurrent connections implement g . The plan units and the state units are the input units. The network takes a plan as input, i.e., a distributed pattern of activation over the plan units, and produces a sequence of actions over the output units.

The Jordan-networks have full connectivity between input, hidden and output layer and one-to-one connectivity from the hidden or output units to the linear state units. Feedback links have a fixed weight (1.0) and their sole purpose is to copy a hidden or output unit activation pattern to the state units. In other words, the activation pattern over the state units at time step t equals the hidden or output unit activation pattern at $t - 1$.

A modification of this approach leads to Elman-networks: *(The) network is augmented at the input level by additional units; call(ed) context units (previously called state units). These units are also "hidden" in the sense that they interact exclusively with other nodes internal to the network, and not the outside world* (Elman, 1989).

The activation patterns of the hidden units at time step t are copied onto the context units and presented with the input at the next time step (see Figure 1): *Thus, the hidden units have the task of mapping both an external input, and also the previous internal state of some desired output. Because the patterns on the hidden units are saved as context, the hidden units must accomplish this mapping and at the same time develop representations which are useful encodings of the temporal properties of the sequential input. Thus, the internal representations that develop are sensitive to temporal context; the effect of time is implicit in these internal states* (Elman, 1989).

2.2 The NECAS recurrent network

The architecture of the recurrent network used in our application is similar to simple Elman-networks. It uses two hidden layers and copy-back links from the first hidden layer to the state units (similar to Elman-Networks). The first hidden layer units, the state units plus the output units are all linear units. There is full connectivity in feedforward direction, but fixed one-to-one connections are used between the first and second

hidden layer. Also, there is fixed one-to-one connectivity between the first hidden layer and the state units.

The use of linear units as part of the first hidden layer allows to preserve input information and to copy the hidden unit activation to the state units before the outputs are squashed by the sigmoidal output function. Without this additional hidden layer, it was not possible during learning to reduce the sum-squared error to a minimum.

The input layer (i.e., the plan units of the NECAS recurrent network) consists of sets of units which represent information from the incoming message (see Figure 1). In detail, there are 8 units for the representation of the account name, 8 units for the coding of the domain, (for instance "cs"), 8 units for the representation of the location (such as "berkeley") and 4 units for the coding of the network (for instance "edu" as part of the Arpa net). Due to the binary input coding, we use 8 bits for the representation of the account name, domain and location and 4 bits for the coding of the network. Furthermore, we currently use 16 units for the representation of the subject (the first non-function word in the subject line is used). A random bit vector represents an individual account name, location, network etc.; it is guaranteed that a bit vector is used only once in the training and test set. The random bit vectors are retrieved from hash tables that are used and updated by the NECAS system. New random bit vectors are generated, stored and used when new information comes in.

Note that we use the recurrent network for sequence production and that the network starts with random initial weights. Due to the fixed plan vector, the network has to use the state information to adjust weights. This training task is difficult, in particular during early periods of the training due to the erroneous activation patterns over the state units. Therefore, the network must have a significant number of hidden layer and state units.

The initial experiments were done with a varying sizes of the input and output layer (44-56 units for account name, domain, location, network and subject in the input layer and 4 or 20 units in the output layer). Good results are reported from experiments with the following configuration: first and second hidden layer as well as the group of state units have 100 units each, the output layer contains 20 units and the plan is a 44 bit vector (Diederich et al., 1992).

2.3 The recurrent cascade correlation architecture

Fahlman (1991) introduced the Recurrent Cascade Correlation Architecture (RCC). Instead of only adjusting the weights of a fixed network topology, the idea is to start with a minimal network and to add hidden units as necessary.

So the initial network starts with no hidden units, and only the weights to the outputs are trained. If this is not satisfying, a new hidden unit has to be added.

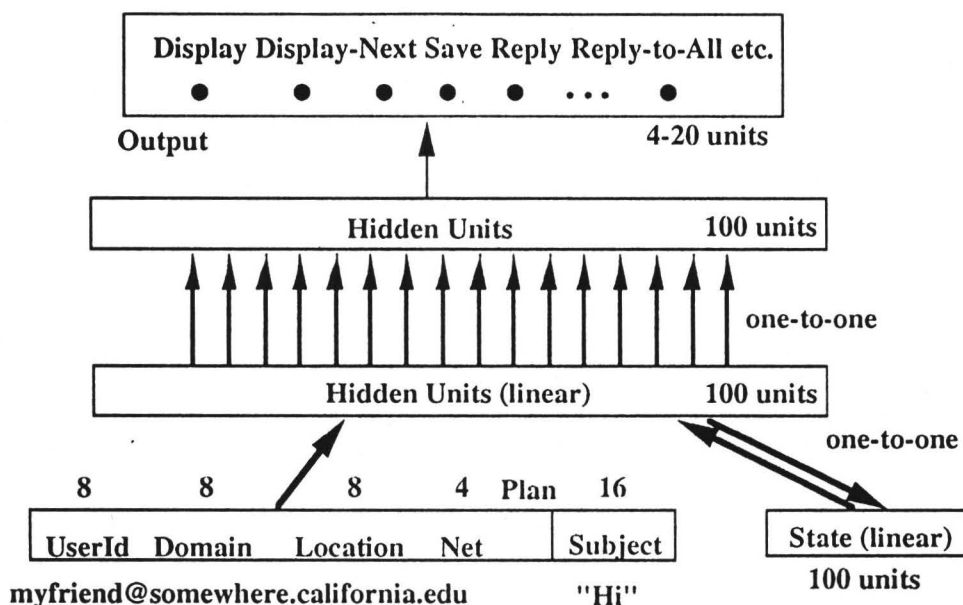


Figure 1: The architecture of the NECAS recurrent network.

The learning and network construction process works as follows: The algorithm starts with a set of new units, called candidate pool. These units have randomly initialized, weighted connections from the input nodes and all the hidden units already present in the network. At that time their outputs are unconnected. Then, their incoming links are trained on the training set and adjusted in order to maximize the correlation between the candidates outputs and the remaining error (Fahlman, 1991). When there is no improvement, this process stops and the candidate with the best "correlation score" is added permanently. Its weights are frozen now, which means that this node becomes a new unique feature detector in the network. In order to integrate this new hidden unit, the weights of the output layer are trained for multiple epochs (Fahlman, 1991). This scheme of adding hidden units and retraining the output layers weights is repeated until success (when the error falls below a predefined threshold or no bit errors are found) or a maximum number of hidden units is reached (which means failure). Fahlman points out that this learning algorithm works like a greedy algorithm because the network is extended stepwise until a solution is found.

Note that every new hidden unit adds a new hidden layer to the network, because it is connected to all previous hidden units. Recurrence is achieved by adding self-recurrent links to all hidden units. So at time step $t + 1$ the activation of time step t is fed back to the hidden unit. Such a constructive learning algorithm is expected to build a near minimal network for a given problem, because every hidden unit serves as a unique feature detector, and no a priori and maybe inappropriate guesses about the size of hidden layers have to be made.

In our experiments, we used the same coding for the inputs and outputs as for the NECAS RNs in section 2.1.

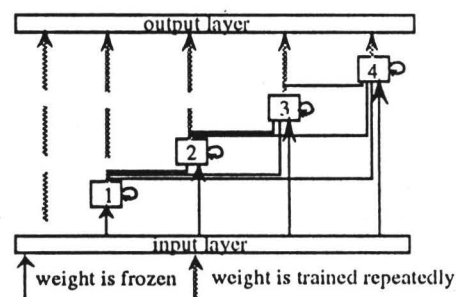


Figure 2: The Recurrent Cascade Correlation Architecture. Example with 4 hidden units.

2.4 The data set

We use artificially generated and natural data (extracted by observing real users) for training and test purposes. The artificial training set is used for evaluation purposes, in order to test various network architectures. Also, because of the size of the NECAS recurrent network, training has to be done with sufficiently large training sets.

An artificial training set was constructed by extracting addresses from the Usenet and defining a number of explicit rules for typical user responses. These rules are used to generate an artificial data set. Six of these rules are "useful heuristics"; two are response patterns to individual senders. The rules encode response tendencies that a real e-mail user might have.

For example, a "useful heuristic" is that messages from sender at well-known research labs and universities are more important than messages from unknown locations. Therefore, the hypothetical user, who is simulated by the artificial data set, responds with long sequences of e-mail actions to messages from "important"

Rule	Number of Sequences	Length (no. of patterns)
1	25	2
2	25	2
3	50	6
4	75	4
5	100	4
6	75	5
7	50	6
8	25	2
9	75	3

Table 1: The rules used for generation of the artificial data set

Position $i =$	# of pattern at position i
1	500
2	500
3	425
4	350
5	175
6	100

Table 2: The number of training patterns per position in a sequence

locations. 550 sequences with a total of 2050 input-output patterns were generated by use of these rules. The length of an individual sequence ranged from two to six patterns (the end of a sequence was indicated by a special marker).

The following two rules are examples for the type of rules used to generate the artificial training set:

1. If a message comes from the location "berkeley," then "display" the message, "save" the message and "reply" to the message.
2. If a message comes from an "administration" (domain) at the location "berkeley," then "display" the message, "save" the message, "save the message in a special folder" and "reply" to the message.

Table 1 shows the structure of the training set. We also counted the number of training patterns that occurred in a certain position inside of a sequence (Table 2).

3 The evaluation method

An important question is: How large will the RCC network be, given the same data set as for the fixed NECAS RN? In order to answer this question, we used the following method: First, we run RCC on the data set until the error was significantly reduced (we assume

a rms-error¹ below 0.1 indicates success) and record the number of hidden units. We built up a NECAS RN with the same number of hidden units and train the network as well. Because RCC has the advantage of shortcuts from inputs to outputs, we have repeated the training session with a modified NECAS RN, where shortcuts from inputs to outputs are part of the network. In an additional experiment, we replaced the linear activation functions of the outputs with sigmoidal functions.

For RCC we used sigmoid unit types (standard range from 0.0 to 1.0), and a sigmoid prime offset of 0.1. The RNs were trained with a small learning rate (0.0005), a momentum of 0.9 or no momentum and a negative bias (-2.5).

Another objective is the evaluation of the sequence production process. Therefore, we use a special kind of error report for all experiments. First we divide the data set p into six sets $p_1, \dots, p_6$. p_i contains all the training patterns which appear in their sequences at position i . Then for each epoch we sum up the squared error for all pattern in p_i , averaged over the size of p_i , resulting in a position error e_i :

$$e_i = \frac{\sum_{p_j=1}^{p_j=n} \sum_{k=1}^n (o_k(p_j) - t_k(p_j))^2}{|p|}$$

where $o_k(p)$ is the activation of output k for pattern p , $t_k(p)$ is the corresponding teaching input and n is the number of output units. This position error allows to show how good the network performs on sequence production.

4 Results

RCC was able to learn the artificial data set in about 4300 epoches. The error decreased continuously as new hidden units were added. The behavior of the NECAS RNs were different. In general, the rms-error was reduced very fast to 0.3, but got stuck above 0.2. A rms-error of 0.26, as computed for one NECAS RN, means a total of 304 output bit errors² over the whole training set. For the RCC network, we counted 31 bit errors after completion of the training. Figure 5 shows that NECAS RNs with linear outputs give slightly better results than networks with sigmoidal outputs. An important detail is that there is only a small difference in performance if shortcuts from inputs to outputs are used. One should expect that networks with these shortcuts are more powerful, so the interpretation is that 43 hidden units are not sufficient. Because we got good results with 100 hidden units (our initial guess),

¹The rms-error is defined by

$$rms = \sqrt{\frac{\sum_{p_j=1}^{p_j=n} \sum_{k=1}^n (t_k(p_j) - o_k(p_j))^2}{|p|}}$$

²An output is counted as erroneous, if its activation deviates from the teaching input by more than 0.3.

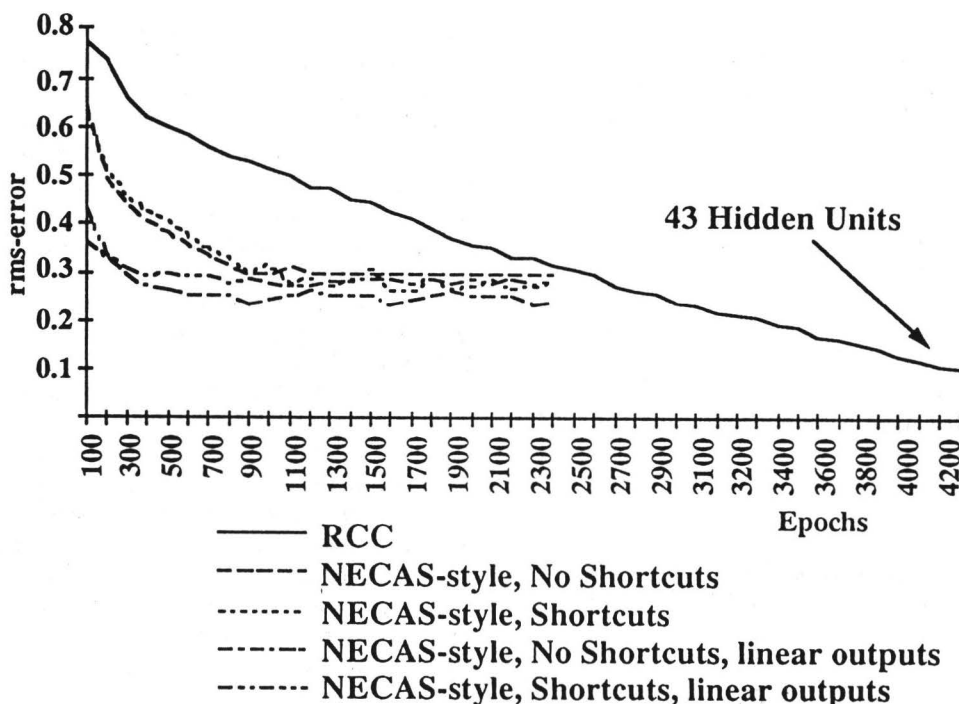


Figure 3: Comparison of the tested networks. Y-axis shows the rms-error over one epoch, x-axis shows the trainings epochs. The NECAS RNs were stopped after 2300 epochs because no further error reduction was observed. Four different SRNs were tested: "NECAS-style" with and without shortcuts, as well as with linear and sigmoidal outputs. In addition, the performance of the RCC-network is shown, which finally has 43 hidden units.

we expect that the optimal SRN for our data set must have about 100 hidden units.

In summary: RCC is able to build up smaller networks compared to the SRNs we tested. Even though more training epochs are necessary, RCC was faster, for the following reasons:

- In backpropagation, each presentation of a training case requires a forward pass and a backward pass through all layers in the network. RCC can train single layers.
- In RCC, many training epochs run while the network is small compared to its final size.

In our simulations, RCC required ten times less computation time than the NECAS RN, for the same number of epochs.

Figure 4 to 6 show the position errors for the compared networks. At the first glance these results are very unexpected. A first assumption is that the error increases with the length of a sequence. But while the error for patterns in the first and the last position is decreased rapidly, the error for the other positions is only slowly reduced. The reason is the structure of the data set. Only 18.1% of all sequences have the maximum length of six patterns. But there are 325 "medium-sized" sequences (that is: 2-4 patterns) which were generated using six rules. This means that the major part of the data set consists of short sequences distributed over many classes.

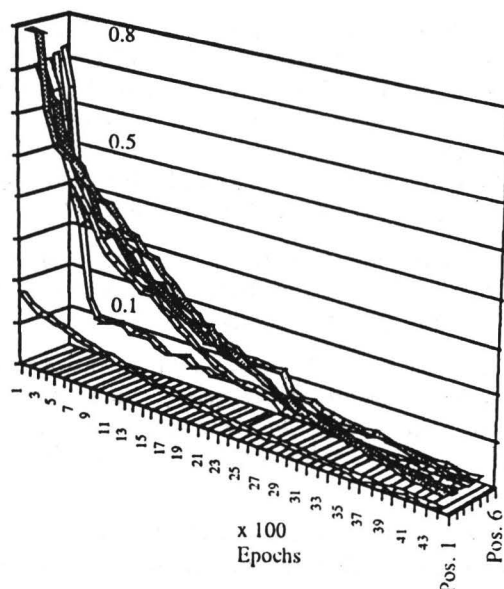


Figure 4: Position error for the pattern sets $p_1, \dots, p_6$ using Recurrent Cascade Correlation. X-axis shows trainings epochs (in 100 epochs), y-axis the position error e_i .

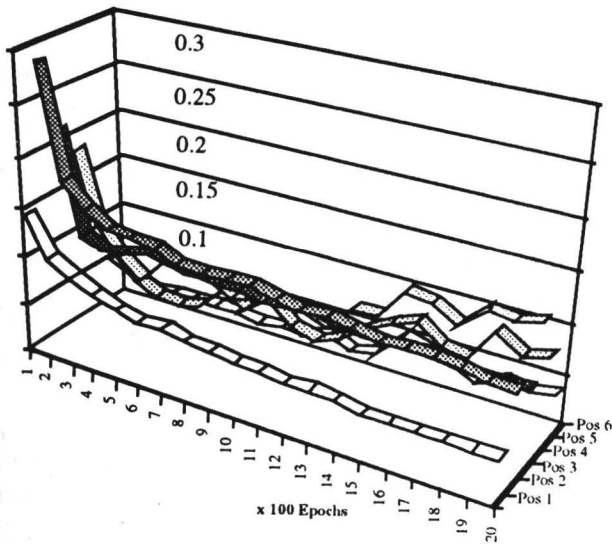


Figure 5: Position error for a NECAS RN with 43 hidden units, shortcuts from the inputs to outputs and linear output units.

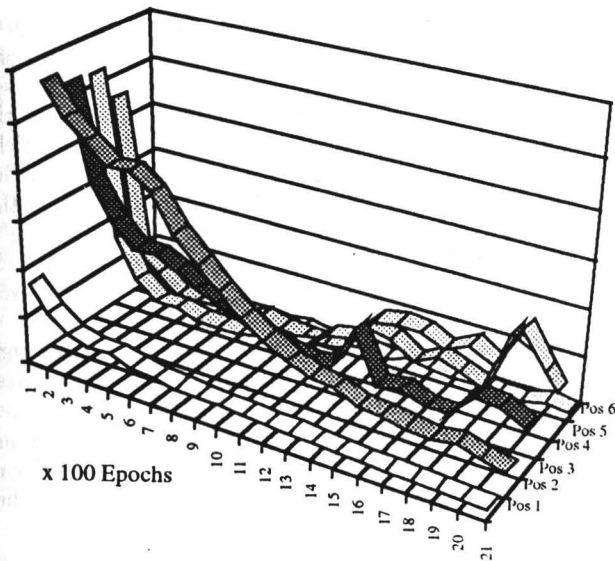


Figure 6: Position error for a NECAS RN with 43 hidden units, shortcuts from inputs to outputs and sigmoid output units.

5 Discussion

To find the right structure and size of a neural network is often difficult and a non-trivial task. Automating this process using incremental algorithms (in our case RCC) is an alternative. Our experiments are a first attempt to determine the effectiveness of RCC in comparison to SRNs on a real-world problem. Our results show that RCC is more efficient. Of particular interest for our application is the ability to generalize from unknown or noisy data. This will definitely be the next step for future research. Even though Fahlman (Fahlman, 1991) reports superior generalization for his benchmark tests, we will have to verify this for our own data set (for the non-recurrent version CCA there exist results that neglect the superiority (Yang & Honavar, 1991)). Another weak point of RCC is the lack of a theoretical model for this architecture and the large number of parameters that control RCC (and often their choice is left to intuition).

6 Acknowledgments

The architecture of the NECAS recurrent network is partially based on suggestions by Andreas Weigend. The use of artificial data sets for test purposes was suggested by David Rumelhart. We would like to thank C. Doherty for making the code for the RCC simulator available.

References

- [1] Diederich, J.; Thuemmel, A.; Bartels, E.: Recurrent and Feedforward Networks for Human-Computer Interaction. In: Neumann, B. (Ed.): ECAI-92, 10th European Conference on Artificial Intelligence. Chichester: John Wiley & Sons 1992, 206-207
- [2] Elman, J.L.: Finding Structure in Time. *Cognitive Science* 14 (1990) 179-211
- [3] Fahlman, S.E.: The Recurrent Cascade-Correlation Architecture. Pittsburgh, Penn.: Carnegie-Mellon University (CMU), May 1991 = CMU-CS-91-110
- [4] Jordan, M.L.: Attractor dynamics and parallelism in a connectionist sequential machine. Proc. of the 8th conference of the Cognitive Science Society 1986. Hillsdale, N.J.: LEA 1986. Reprinted in: Diederich, J. (Ed.): Artificial Neural Networks: Concept Learning. IEEE Computer Society Neural Network Technology Series, 112-127, Los Alamitos, Ca. 1990
- [5] Yang, J. & Honavar, V.: Experiments with the Cascade-Correlation Algorithm. Dep. of Computer Sciences, Iowa State University, IA, USA., 1991 = Tech. Report #91-16