Hill Climbing in Recurrent Neural Networks for Learning the $a^n b^n c^n$ Language

Stephan Chalup
Machine Learning Research Centre
School of Computing Science, FIT
Queensland University of Technology
Brisbane, Australia
chalup@fit.qut.edu.au

Alan D. Blair*
Department of Computer Science
and Electrical Engineering
University of Queensland
Brisbane, Australia
blair@csee.uq.edu.au

Abstract

A simple recurrent neural network is trained on a one-step look ahead prediction task for symbol sequences of the context-sensitive $a^n b^n c^n$ language. Using an evolutionary hill climbing strategy for incremental learning the network learns to predict sequences of strings up to depth n = 12. Experiments and the algorithms used are described. The activation of the hidden units of the trained network is displayed in a 3-D graph and analysed.

1 Introduction

It has been known for some time that recurrent neural networks can be trained to recognise or predict formal languages. (Siegelmann & Sontag, 1992) showed that neural networks are capable of universal computation, and therefore able in principle to process any recursive language (although, if we demand robustness to noise, they are limited to the regular languages, see Casey, 1996, Maass & Orponen, 1997). However, their ability to *learn* language processing tasks is still being explored.

Several studies have demonstrated that first and second order recurrent networks can be trained to induce simple regular languages from examples (Pollack, 1991, Giles et al., 1992).

Wiles and Elman (1995) showed how networks can be trained by backpropagation to *predict* the context-free language $a^n b^n$. Others have proposed hand-crafted networks for this task (Hoelldobler, 1997) or for *recognising* the $a^n b^n$ language and the context-sensitive language $a^n b^n c^n$ (Finn, 1998), and for predicting other context-sensitive languages (Steijvers & Grünwald, 1996) without addressing learning issues.

Later studies of learning have revealed that backpropagation tends to encounter instabilities when training on the $a^n b^n$ task (Rodriguez et al., 1999) and that evolutionary algorithms may be able to avoid some of these instabilities (Tonkes et al., 1998). In the present work, we extend this evolutionary approach to the task of predicting the language $a^n b^n c^n$.

In sections 2 and 3 we describe the neural network architecture and the basic evolutionary hill climbing algorithm which was used to train the networks. The $a^n b^n c^n$ task and the incremental learning strategy for this task are topic of section 4 and 5. Experiments are described in section 6 and an analysis of the resulting neural network is given in section 7. An algorithm which generalises our experimental approach is proposed in section 8 and section 9 concludes with some discussion and an outlook for future research.

2 Neural Network

We used a simple recurrent neural network architecture (Elman, 1990) with three units in each layer, as shown in Figure 1.

The most common activation function for neural networks is the sigmoid function. This function was employed in experiments of (Wiles & Elman, 1995). In (Hoelldobler et al., 1997) the approximately linear part of the

*Current address: Department of Computer Science, University of Melbourne, 3052 Australia

sigmoid function was used to design a network for the $a^n b^n$ task. In our present work a hyperbolic tangent function is used. It has the same shape as the standard sigmoid function but is translated so that it is rotationally symmetric about the origin.

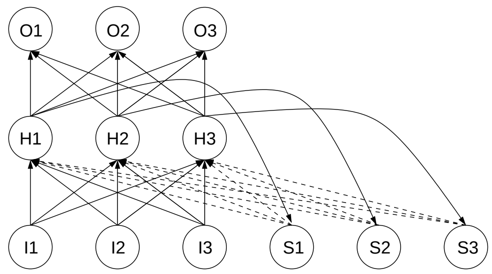


Figure 1. Simple recurrent network with input units I1-I3, hidden units H1-H3, state units S1-S3 and output units O1-O3. Bended arrows are fixed one to one copy connections. Dashed arrows connect the state layer to the hidden layer and operate with a delay of one time step.

3 Evolutionary Hill Climbing

Backpropagation is the most studied and used training algorithm for artificial neural networks ever since (Rumelhart et al., 1986). (Wiles & Elman, 1995) employed it for training a simple recurrent network on the $a^n b^n$ task. Evolutionary hill climbing (see Table 1) is an alternative training algorithm.

While backpropagation is regarded as faster and more sophisticated in general, hill climbing has some advantages, too. It can train networks with non-differentiable activation functions; it is easy to implement and it offers a high degree of flexibility in the design of the training strategy. This flexibility is due to an error or fitness function which may be non-differentiable.

Hill climbing uses batch learning, i.e. the decision of a weight update is made at the end of each epoch on the basis of the training error which is calculated over all patterns of that epoch. This is in contrast to on-line learning where weight updates can be made after each single pattern throughout the epoch.

```
Evaluate network with initial weights W_champ
==> error_champ

WHILE((error_goal <= error_champ) and
    (counter <= nEpochs))
    FOR all weights of the neural net
        ΔW_mutant <- random number * stepsize
        W_mutant <- W_champ + ΔW_mutant
    end FOR
    Evaluate network given by W_mutant
    ==> error_mutant
    IF(error_mutant < error_champ)
      W_champ <- W_mutant
      error_champ <- error_mutant
    end IF
    counter <- counter + 1
end WHILE

return error_champ
```

Table 1. Evolutionary Hill Climbing

We trained our networks using an evolutionary hill climbing algorithm whose basic version is displayed in Table 1. We call the weight matrix $W_{champ} = (w_{ij})$ and the corresponding neural network the *champion*. A weight w_{ij} connects unit j to unit i. In our experiment the initial weights were generated from a N(0,0.05) normal distribution. Given the champion W_{champ} the hill climber endeavours to find a "better" matrix by generating a *mutant* matrix $W_{mutant} \leftarrow W_{champ} + \Delta W_{mutant}$ and comparing the champion and the mutant by evaluating the corresponding networks on the training set. The resulting two error values ($error_{champ}$ and $error_{mutant}$) can be compared. The coefficients of the matrix ΔW_{mutant} are randomly generated from a Cauchy distribution. The Cauchy distribution has a similar shape to a normal distribution but it has thicker tails which sometimes is an advantage for hill climbing, see (Chalup & Maire, 1999).

4 The $a^n b^n c^n$ Task

The network is presented with a series of strings $a^n b^n c^n$ one after the other, for varying values of n. The task is to predict the next symbol in the sequence as accurately as possible. The symbols a, b and c are represented by vectors (100), (010) and (001), respectively, for both the observed input and the target output. The next symbol predicted by the network is a, b or c depending on which of the three outputs has highest activation. Since the value of n is not known by the network at the start of a new string, it is impossible for it to predict when the first b will occur. However, once it has "seen" the first b, it is required to predict $n-1$ additional b's, followed by n c's, followed by an indeterminate (but nonzero) number of a's (which form the beginning of the subsequent string).

5 Incremental Learning Strategy

The standard definition of *incremental learning* means that either the dataset or the neural network structure is incrementally enlarged (e.g. by insertion of additional hidden units) during training. (Elman, 1993) used two versions of incremental learning which were entitled: *incremental input* and *incremental memory*. In the incremental input approach simple recurrent networks were trained to learn grammars while the complexity of the sentences in the training data was gradually increased. The training was conducted in five stages, with each stage using a different training set. In the incremental memory approach, the full data set was used but the time window of the simple recurrent network was restricted in the beginning and enlarged during training.

The present study used a form of staged learning which is similar to the incremental input approach of (Elman, 1993). We generated a sequence of ten training sets, each of them corresponding to a stage of increased difficulty in the training. The training set of stage d was the concatenation of $d-1$ strings of the form $a^n b^n c^n$, $2 \leq n \leq d$. For example, the training set for stage 4 consisted of the following sequence of 27 symbols:

$$aabbccaaabbbcccaaaabbbbcccc$$

The training set of the next higher stage contained the same sequence of symbols but with the string *aaaaabbbbbccccc* concatenated at the end.

The error calculation took all symbols of the training sequence except the first b of each string into consideration (see section 4).

At each training stage d the mean squared error was separately calculated for the concatenation of the first $d-1$ strings of the sequence (mse_{Low}) and for the last string (mse_{High}). The fitness of the network at the end of each epoch could then be calculated as a linear combination of both errors:

$$\text{fitness} = \lambda_{\text{Low}} * \text{mse}_{\text{Low}} + \lambda_{\text{High}} * \text{mse}_{\text{High}},$$

with $\lambda_{\text{Low}}, \lambda_{\text{High}} > 0$. Similarly the accuracy of correct prediction was separately calculated for the low and the high part of the string.

After some preliminary tests, we used $\lambda_{\text{Low}} = 1.0$ and $\lambda_{\text{High}} = 0.5$ and decided on the following acceptance rule for each new mutation:

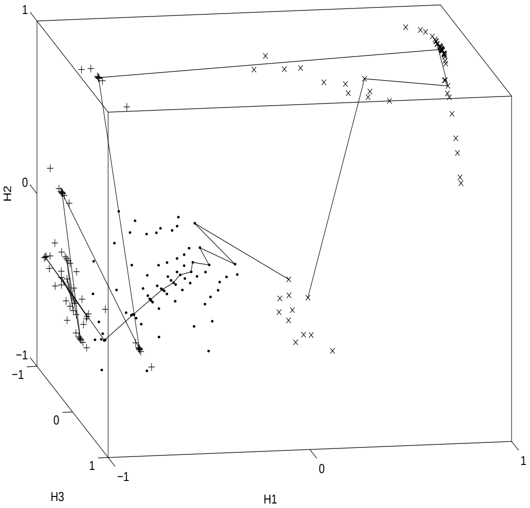
Accept the new mutant if its fitness is higher than that of the champ and if its accuracy on the low part of the string equals 1.0.

An exception was made at the beginning of the training; the data set of the first stage consisted of strings for $n = 2$ and $n = 3$ and the fitness was simply $\text{mse}_{\text{Low}} + \text{mse}_{\text{High}}$.

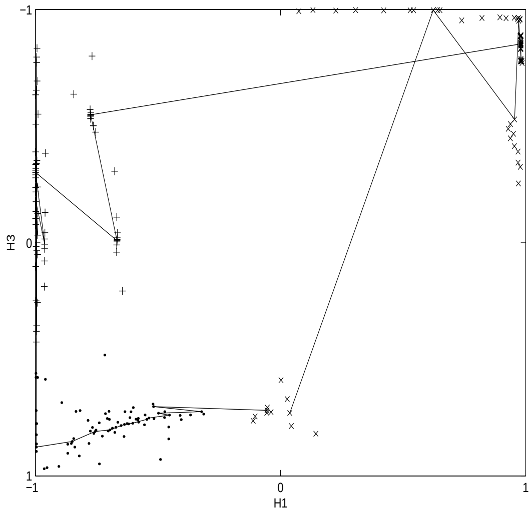
6 Experiments

In preliminary experiments we followed the hill climbing strategy of (Tonkes et al., 1998) to train a simple recurrent network on the $a^n b^n$ task. We were able to obtain similar results when using our implementation of the algorithms.

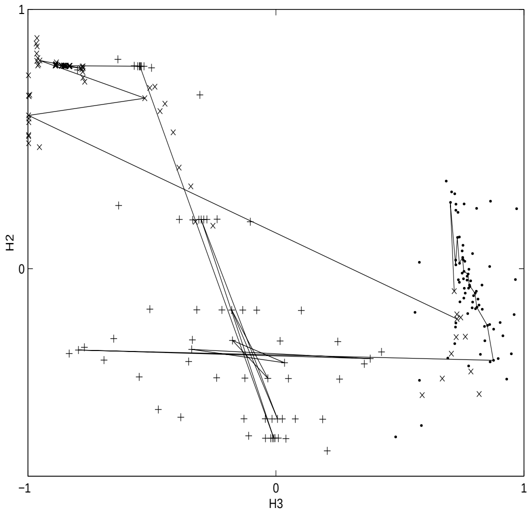
On the $a^n b^n c^n$ task a series of experiments were conducted with different initial weights, seeds and small modifications of the fitness function and the activation functions. In these preliminary tests, we found that most



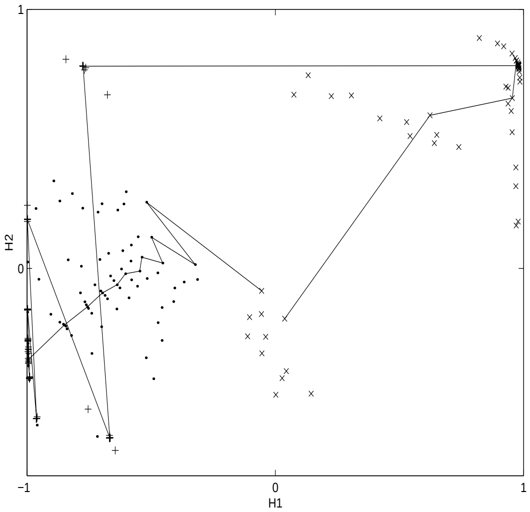
3D view



(a) top view



(b) side view



(c) front view

Figure 2. Trajectory in the 3D hidden unit space, including top view (a), side view (b) and front view (c).

of the networks got stuck in an early stage (below stage 6) of the incremental training. The training strategy seemed to be too rigid for the appropriate attractors and repellers to develop.

The algorithm was then relaxed using two different methods which both were successful in training the network up to depth 10 or 12. First the hill climber was modified to a simulated annealing algorithm with relatively high temperature. The backward steps allowed a form of relaxed training. In the second method we used the standard hill climber and when the training got stuck (in our case at stage 5) we replaced the original training set by a training set in which the order of the patterns was slightly permuted, and retrained the network on this new set. The hill climber was able to reach stage 9 using this training set, after which it got stuck again. We then returned to the original training set, and the network quickly trained up to level 12. The outcome of this training experiment is the network which is analysed in the next section. Finally in section 8 a generalisation of the second training strategy, the Data Juggling Algorithm is proposed.

7 Analysis

Figure 2 shows all the points visited, within the 3-dimensional hidden unit activation space of the network, as it processes the series of strings $a^n b^n c^n$, for $2 \leq n \leq 12$. Activations at which the network predicts an a, b or c are indicated by a '$\times$', '$+$' or '.', respectively. The lines in the figures indicate the path through the activation space as the final string $a^{12}b^{12}c^{12}$ is processed. The way the task is accomplished can be understood by analogy with previously known solutions for the $a^n b^n$ prediction task (Wiles & Elman 1995), which involved a combination of an attractor and a repeller. The network achieved the task by effectively "counting up" the number of a's as it converged to the attractor, and then "counting down" the same number of b's as it diverged from the repeller.

In the present case, the network begins by counting up the number of a's as it converges to an attractor in the top right corner of the hidden unit space (Figure 2(a)). Upon presentation of the first b, the activation shifts to the left side of the space (more clearly visible in Figure 2(b)), where it employs a two-pronged strategy of counting *down* by divergence from a repeller in the H3 dimension, while simultaneously counting *up* by convergence to an attractor in the H2 dimension. The former ensures that the first c is predicted correctly, while the latter prepares for the c's to be counted down by divergence from a new repeller (Figure 2(c)), ready to predict the a at the beginning of the next string.

8 Data Juggling Algorithm

The idea of the Data Juggling Algorithm whose pseudo code is listed in Table 2 is to use the function 'juggle' to permute the order of the symbol strings in the training sequence and to continue training on the modified training set. 'juggle' can be called at any time depending on some conditions, e.g. as soon as the champion has reached a new stage or after a fixed number (nEpochs) of iterations (i.e. when the algorithm got stuck). Using the Data Juggling Algorithm the network learned further up to stage 15. The network generalised to some of the permuted strings perfectly and to most of the others with reasonably high accuracy.

```
WHILE(depth_stage ≤ depth_MAX)
    Evaluate champ
    FOR ce = 1 to nEpochs
        Generate mutant
        Evaluate mutant
        IF(mutant better than champ)
            champ ← mutant
            IF(depth_champ > depth_stage)
                depth_stage ← depth_champ
                break
            end IF
        end IF
    end FOR
    IF(juggling conditions)
        juggle(depth_stage)
    end IF
end WHILE
```

Table 2. Data Juggling Algorithm.

9 Conclusion

We have shown that a neural network using a simple evolutionary algorithm can learn to predict the language $a^n b^n c^n$ with a fixed order and can generalise to other orderings with good accuracy.

Within the Chomskyan framework, the context-sensitive language $a^n b^n c^n$ is considered to be at a distinctly higher level of complexity than the context-free language $a^n b^n$. The fact that a neural network can learn to predict $a^n b^n c^n$, using similar techniques to those employed for $a^n b^n$, provides support to the view that the language complexity classes appropriate for dynamical systems may be different from those developed for symbolic systems. Evolutionary techniques seem to overcome some of the instability issues encountered with backpropagation. In further work we hope to conduct a more comprehensive set of experiments, and to check more thoroughly the generalisation abilities of the network. It is hoped that this study may open the door for application of neural network techniques to a wider variety of languages.

Acknowledgements

Thanks to Frederic Maire, Elizabeth Sklar and Janet Wiles for helpful comments.

References

Casey, M. The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction, *Neural Computation* 8(6), pp. 1135–1178, 1996.

Chalup, S., F. Maire. A study on hill climbing algorithms for neural network training, *Proceedings of the Congress on Evolutionary Computation (CEC'99)*, Washington D.C., pp. 2014–2021, 1999.

Elman, J.L. Finding structure in time, *Cognitive Science* 14, pp. 179–211, 1990.

Elman, J.L. Learning and development in neural networks: The importance of starting small, *Cognition* 48, pp. 71–99, 1993.

Finn, G.D. A recurrent neural network for the language $a^n b^n$: Encoding and decoding binary strings in a neuron's activation function, *Internal Report, Machine Learning Research Centre, Queensland University of Technology*, 1998.

Giles, C.L., C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, Y.C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks, *Neural Computation* 4(3), pp. 393–405, 1992.

Hoelldobler, S., Y. Kalinke, H. Lehmann. Designing a counter: another case study of dynamics and activation landscapes in recurrent networks, *Proceedings of 21st German Conference on Artificial Intelligence*, LNAI 1301, Springer Verlag, pp. 313–324, 1997.

Maass, W., P. Orponen. On the effect of analog noise in discrete-time analog computation, *Proceedings Neural Information Processing Systems*, pp. 218–224, 1996.

Pollack, J. The induction of dynamical recognizers, *Machine Learning* 7, pp. 227–252, 1991.

Rodriguez, P., J. Wiles, J.L. Elman. A recurrent neural network that learns to count, *Connection Science* 11(1), pp. 5–40, 1999.

Rumelhart, D.E., G.E. Hinton & R.J. Williams. Learning representations by back-propagating errors, *Nature* 323, pp. 533–536, 1986.

Siegelmann, H.T., E.D. Sontag. Neural networks with real weights: Analog computational complexity, *Report SYCON-92-95*, 1992.

Steijvers, M., P. Grünwald. A recurrent network that performs a context-sensitive prediction task, *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, Morgan Kauffman, pp. 335–339, 1996.

Tonkes, B., A. Blair, J. Wiles. Inductive bias in context-free language learning, *Proceedings of the Ninth Australian Conference on Neural Networks (ACNN'98)*, Brisbane, pp. 52–56, 1998.

Wiles, J., J. Elman. Learning to count without a counter: a case study of dynamics and activation landscapes in recurrent neural networks, *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pp. 482–487, 1995.