Inductive Bias in Context-Free Language Learning

Brad Tonkes* and Alan Blair* and Janet Wiles*†
*Department of Computer Science and Electrical Engineering
†School of Psychology
University of Queensland
{btonkes,blair,janetw}@it.uq.edu.au

ABSTRACT

Recurrent neural networks are capable of learning context-free tasks, however learning performance is unsatisfactory. We investigate the effect of biasing learning towards finding a solution to a context-free prediction task. The first series of simulations fixes various sets of weights of the network to values found in a successful network, limiting the search space of the backpropagation through time learning algorithm. We find that fixing similar sets of weights can have very different effects on learning performance.

The second series of simulations employs an evolutionary hill-climbing algorithm with an error measure that more closely resembles the performance measure. We find that under these conditions, the network finds different solutions to those found by backpropagation, and is even biased towards finding these solutions. An unexpected result is that the hill-climbing algorithm is capable of generalisation. The two simulations serve to highlight that seemingly similar biases can have opposite effects on learning.

1. Introduction

While a large body of research has focused on the relationship between recurrent neural networks (RNNs) and deterministic finite automata (DFA) (see for example [2, 4]), recent results have shown RNNs capable of learning context-free languages [5, 7] and have suggested that RNNs may have a bias towards inducing non-regular languages [1]. This result gives us a timely warning that when we consider language induction, we should remember the types of biases that are inherent in the learning situation. The induced language is related not only to the training language but also to the induction biases of the system.

Sources of inductive bias can be broadly categorised into three classes: data, architecture and the learning algorithm. There is also some bias as a result of the interaction between these sources. Adding one source of bias can effect what other sorts of biases may be added. For example, in the second series of simulations, changing the error measure used for learning effects the type of learning algorithms that can be applied. A way of adding bias to language induction in RNNs is the choice of training data. Training data may be manipulated by changing the distribution or order of strings in the training set. A number of results indicate that it is advisable to bias RNNs by using shorter strings early in training [3].

Ideally, adding biases to a system should improve the performance of learning. Improvement may be realised by an increase in *efficiency*: the amount of training required to find a solution; *reliability*: the regularity with which a successful network is induced; or *consistency*: whether or not continued training improves performance on the task.

This study investigates the effect of adding biases to a network learning a context-free task: prediction of the next letter in a sequence from the language $a^n b^n$. The solution to this task is well understood [5, 7]. Learning this task with backpropagation through time appears to be inefficient, unreliable and particularly inconsistent [6]. With this previous result in mind we consider two ways of biasing our system and examine the resulting effects on learning performance.

2. Backpropagation Through Time

2.1. Issues

The standard backpropagation through time algorithm (BPTT) [8] is not particularly effective at inducing a network that performs the $a^n b^n$ prediction task. It typically fails to find a solution in a reasonable amount of time [7], and any solution that is found is lost with further training [6]. Consequently, we consider mechanisms to bias learning towards finding desirable solutions.

One possible way of adding bias to learning is to simplify the network so that BPTT does not need to fit as many parameters. Since the architecture we are using is of close to minimal size, we cannot simply remove weights. An alternative is to fix some of the weights by setting them to values found in a successful network and not modifying them during training. Rather than fixing random collections of weights, we choose specific groups that contribute to some property of the dynamics.

By fixing collections of weights, we reduce the size of the space that BPTT searches. Ideally, there should be some set of weights to fix that restricts BPTT to searching along smooth directions of the error surface, improving the consistency and efficiency of learning. Additionally, restricting the search space of BPTT should not prevent it from finding a solution — reliability should also be improved.

2.2. Simulation

One network was trained with BPTT on the $a^n b^n$ prediction task, with n varying from 1 to 10. At a point where the network successfully performed the task for all of the training strings and generalised to $n = 11$ and $n = 12$, the weights of the network were saved. Subsequent networks had some of their weights fixed to these values, shown in figure 1. This network displayed dynamics similar to those shown in figure 2 (bottom).

Twelve different sets of weights were selected to be fixed during training. We report the seven most interesting cases. These seven cases set the following weights to values found in a successful network, and fix them at these values during learning.

1. The weights into and between the hidden units, and hidden unit biases.
2. The hidden to output weights, and output biases
   (a) only;
   (b) and recurrent weights;
   (c) and mutual-recurrent weights[1];
   (d) and self-recurrent weights.
3. The recurrent weights.
4. The non-recurrent weights, and all biases.

For each of these seven conditions, 25 or 50 networks were trained with a sequence of 10000 strings from the language $a^n b^n$, with more shorter strings than longer strings. Training strings were between 2 ($n = 1$) and 20 ($n = 10$) characters long. After each string in the sequence had been presented, the weights of the network were saved and the performance of the network tested for n varying from 1 to 12. This process yields 10000 test points during training.

[1] From the first hidden unit to the second, and vice-versa.

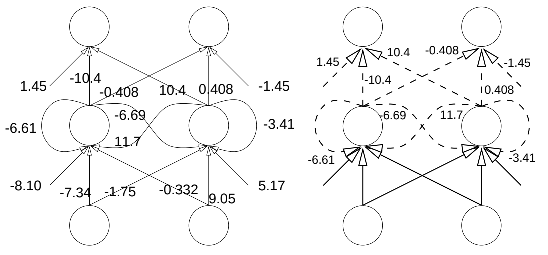

Fig. 1: The weights of the successful network and the training condition where the output unit weights and biases and the recurrent weights are fixed.

The weights of the network that were not fixed were initialised to random values between -1 and 1. A learning rate of 0.05 and a momentum term of 0.3 were applied during learning. Nine copies of the hidden units were maintained by the algorithm. These parameters are identical to those used in [6].

The consistency of learning is characterised by the number of these tests on which the network is able to predict all 12 test strings. Similarly, we measure the efficiency of learning by the number of strings presented before the first successful test. The number of networks that ever give a positive test result indicates the reliability of learning.

2.3. Results

Table 1 shows how successful BPTT was under each of the fixed weight conditions, as well as the results reported in [6] when no weights were fixed.

Fixed Weights	Nets	% Success (Reliability)	Correct (Consist.)	Fastest (Effic.)
None	100	13	20	≈ 5000
Hidden Unit	50	44	44	70
Output Unit	50	52	85	2092
& Rec.	50	0	—	—
& Mut-Rec.	25	0	—	—
& Self-Rec.	25	96	23	540
Non-Rec.	25	48	94	5440
Recurrent	50	0	—	—

Table: 1: Results of simulations. Listed are the tied weights, the number of networks trained, the percentage of the trained networks that were successful, the maximum number of times a network tested correctly, and the minimum number of strings required before a network tested correctly.

Not surprisingly, the case where all hidden unit weights were fixed (case 1) found solutions very quickly (70 compared to $\approx$5000 strings). In this situation only the output hyperplanes are learned. However, the network was unable to maintain the solution, with the hyperplane often misclassifying the longer strings. We believe that this result is a consequence of the distribution of strings, with the greater proportion of shorter strings tending to pull the hyperplane closer to the "b" saddle point.

The 'dual' to this network is the one with only the weights into the output units fixed (case 2a). The networks trained under these conditions were more successful and still significantly faster than the "standard" case. Learning with these biases was comparably reliable and more consistent than the 'dual' condition, though not as efficient. Given this success, we expected that fixing the output hyperplanes as well as some of the recurrent weights would have improved the performance of BPTT further. However, when in addition to the output hyperplanes, all of the recurrent weights (case 2b) or the mutual recurrent weights (case 2c) were fixed, BPTT completely failed to find a correct network. Incongruously, fixing the output unit weights and the self-recurrent weights (case 2d) resulted in a 96% success rate, as well as significantly boosting the efficiency of learning, although reducing consistency.

Fixing the non-recurrent weights (case 3) of the network significantly improved reliability, but learning was neither consistent nor more efficient than the standard case. No network with all recurrent weights fixed (case 4) found a solution.

3. Evolutionary Hill Climbing

3.1. Issues

The first simulation focused on biasing learning by restricting the search space. In a second series of simulations, we consider an alternative method of adding bias by changing the error measure used during learning. BPTT performs pseudo-gradient descent on the mean squared error (MSE) of the network, whereas the performance of the network is evaluated by the length of strings that are correctly predicted based on a 0.5 output threshold. Consequently, the next series of simulations involve choosing an error measure that is commensurate with our method of evaluation, based on the number of correct strings (NCS). The NCS for a network is the largest N such that the network can successfully predict all strings from $n = 1$ to $n = N$.

We used a hill-climbing evolutionary algorithm (EA) with a two-tier error measure to learn the task. The first discrimination made between two networks is the NCS measure, with MSE used only as a secondary discrimination between networks. The advantage of this algorithm is improved consistency. Since the EA hill-climbs on NCS, it also hill-climbs on our performance measure.

One possible drawback to using an evolutionary algorithm is the efficiency of the approach. Whereas BPTT performs a directed search of the space based on error, an EA searches in a randomised fashion. For this reason we expected the EA to require a considerable amount of time to find solutions.

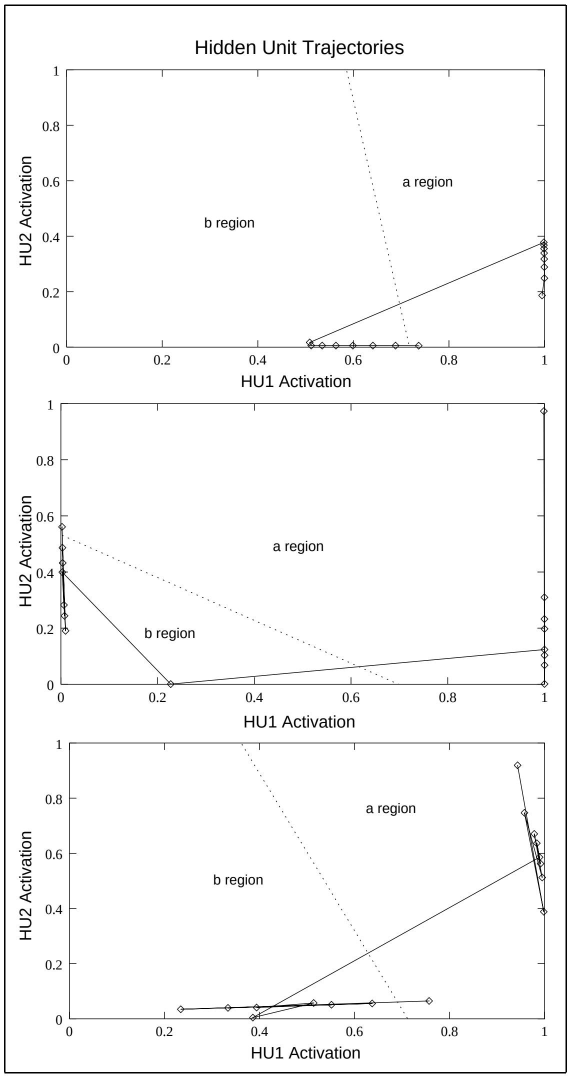

Fig. 2: Hidden unit trajectories and output hyperplane for a^8b^8 for the three solution classes. Top: monotonic solution with signature (2,0); middle: exotic oscillating solution with signature (1,1); bottom: typical oscillating solution with signature (0,2).

3.2. Simulation

The network architecture was slightly altered for this simulation. In order to reduce redundant weights in the network that would have slowed the algorithm, only one output unit was used. The initial network is generated with random weights chosen from a gaussian distribution with standard deviation 0.1.

At each generation, a new network is generated from the current network by adding to each weight a gaussian random variable with standard deviation

0.01. Suppose the current network can correctly predict[2] all strings $a^n b^n$ for $1 \leq n \leq N$, but fails for $n = N+1$. If the new network can correctly predict $a^n b^n$ for $1 \leq n \leq M$, with $M > N$, then replace the current network with the new one. If the new network can correctly predict $a^n b^n$ for $1 \leq n \leq N$, calculate the mean squared error it accrues while processing $a^{N+1} b^{N+1}$. If this error is lower for the new network than the current network, replace the current network with the new one.

In preliminary training, we ran this algorithm three times, and found that in each case it was able to find a solution achieving depth greater than 30. Once the depth reaches about 15, it typically begins to jump by 2 or 3 at a time. This phenomenon can be considered as a form of generalisation. Although the network is being evolved to the next largest string, it is generalising to even longer strings.

One of these networks found an oscillating solution (figure 2 bottom) of the kind commonly produced by backpropagation [7]. One found a monotonic solution which counts by using two attractors rather than an attractor and a saddle point (figure 2 top). Solutions of this kind had been found by backpropagation, but typically were unable to generalise to longer strings [5]. It is interesting, therefore, that this monotonic network demonstrated considerable capacity for generalisation.

The third run (actually a slight variant of the above algorithm) produced an unusual solution of a kind that had not previously been observed (figure 2 middle). In this network, "counting" of both the *a*s and *b*s is performed by the same hidden unit, with the other unit used to "switch mode".

3.2.1. Actual Trials

To test the reliability of this algorithm, we then performed 50 more runs for 100,000 generations each.[3] The average number of strings tested per generation was 5.7. The average length of each string tested was 9.3.

A run was defined as "successful" if it achieved depth 12 within 100,000 generations.

The weight sets produced by this algorithm can be classified according to the *signature* (p, q) of the recurrent weights, where p and q are the number of positive and negative self-weights, respectively. Networks with signature $(0, 2)$ always exhibited oscillating behaviour, while those with signature $(2, 0)$ were monotonic in nature. Of those with signature $(1, 1)$ about half were oscillating and half monotonic. The exotic solution of figure 2 (middle) also had signature $(1, 1)$.

[2]Excluding the non-deterministic step.

[3]This number of generations was selected to be comparable to the number of weight updates made by BPTT on 10000 strings.

3.3. Results

Eighteen of fifty runs found networks with NCS $\geq$ 12. Of the 50 final networks, 5 failed to predict any strings correctly, consequently did not produce any output and were not classified. The performance of the EA is summarised in table 2.

signature	unsuccessful	successful	total
(2,0)	2	11	13
(0,2)	15	6	21
(1,1)	10	1	11
—	5	0	5
total	32	18	50

Table: 2: Success of the evolutionary algorithm at inducing correct networks. The algorithms is judged to be successful if it induces a network that can correctly perform the prediction task for n varying from 1 to 12. For the purposes of this table, the signature of a network is taken from the network at the final generation.

As table 2 shows, networks with signature $(2, 0)$ were most likely to be successful, those with signature $(1, 1)$ least likely. Figure 3 plots the average depth as a function of generation, grouped by signature.

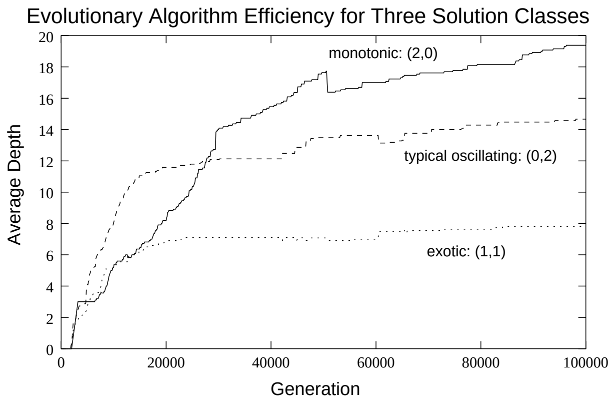

Fig. 3: Efficiency of the algorithm at inducing correct networks for each of the three solution classes. Note that the plots are not always increasing because the signature occasionally changes during the course of a run.

4. Discussion

The two groups of simulations explored the effects of adding bias on learning performance on the $a^n b^n$ prediction task. The first series of simulations biased learning by restricting the search space in different ways and found that there was no simple relationship between the fixed weights and learning performance. When the output unit weights and biases were fixed, BPTT found a solution around 50% of the time (compared with 13% in the unrestricted case). Additionally fixing the self-recurrent weights resulted in near-perfect reliability, but when all recurrent weights were fixed, BPTT never found a solution. The dramatic change in performance may be a result of preventing BPTT from searching in necessary regions of the space. Overall, the performance of BPTT proved to be very sensitive to restrictions to the search space.

The performance of the evolutionary hill-climbing algorithm was somewhat surprising. Not only did the algorithm find solutions within a number of weight updates comparable to BPTT, but the best trials found solutions that could perform the task for very long strings — indeed the best solutions we have found yet (up to $n = 53$). Furthermore, the evolutionary algorithm found not only the monotonic solution that BPTT does not find very easily [5], but a solution that BPTT has never found. In contrast to BPTT, the evolutionary algorithm seems to have a preference for monotonic solutions, inducing more successful networks of this type than successful oscillating networks.

The most surprising finding of the simulations was the way in which the evolutionary algorithm generalised. Networks with both oscillating and monotonic dynamics often increased their depth of processing by more than one string in a single generation. In one case a single update improved the performance of a monotonic network from $n = 33$ to $n = 45$. Since the error measure used by the algorithm is aimed specifically at only the next longest string, there seems to be some bias inherent in the network towards inducing longer strings.

5. Conclusions

Not surprisingly, our simulations have shown that adding bias to a system has an impact on learning. However the extent of this impact is somewhat surprising. The first series of simulations demonstrated that applying two seemingly similar biases could result in near-perfect reliability or failure to find any solutions. From this we gather that learning is particularly sensitive to biases.

Another unexpected result was that the hill-climbing algorithm induced successful monotonic networks unlike BPTT. Indeed, the EA actually appeared biased towards finding this solution, with more successful networks displaying monotonic rather than oscillating dynamics. In light of this result, one can see how RNNs may be biased towards inducing non-DFA-like dynamics even when trained on regular languages [1].

This possibility is given even more credence by the finding that the evolutionary hill-climbing algorithm was capable of generalisation. The way that generalisation occurs — in a context-free manner — suggests that RNNs may be biased towards finding solutions that are not analogous to DFA computation.

The simulations detailed in this paper are part of work in progress. One of the fixed-weight conditions significantly improved reliability, and others improved efficiency. Furthermore, the EA was able to learn consistently, unlike BPTT. Although no single bias improved the reliability, efficiency and consistency of learning, the individual biases gave significant clues as to what makes an effective bias. Based on these clues we seek biases that improve all three measure of learning performance.

Acknowledgements

This work was supported by an APA to Brad Tonkes, a UQ Postdoctoral Fellowship to Alan Blair, an ARC grant to Janet Wiles, and by the CSEE department.

References

[1] A. Blair and J. Pollack. Analysis of dynamical recognizers. *Neural Computation*, 9(5):1127 – 1142, 1997.

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

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

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

[5] P. Rodriguez, J. Wiles, and J. Elman. A dynamical systems analysis of a recurrent neural network that learns to represent the structure of a simple context-free language. Under review.

[6] B. Tonkes and J. Wiles. Learning a context-free task with a recurrent neural network: An analysis of stability. To appear: *Proceedings of the Fourth Biennial Conference of the Australasian Cognitive Science Society*, 1997.

[7] J. Wiles and J. Elman. Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 482 – 487, Cambridge, MA, 1995. MIT Press.

[8] D. Zipser. Subgrouping reduces compexity and speeds up learning in recurrent networks. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, pages 638 – 641. Morgan Kaufmann Publishers, 1990.