Utilizing Bias to Evolve Recurrent Neural Networks

Edwin D. de Jong and Jordan B. Pollack

Brandeis University
Computer Science Department
Waltham, MA 02454, USA
{edwin, pollack}@cs.brandeis.edu

Abstract

Since architectures and weights for recurrent neural networks are difficult to design, evolutionary methods may be applied to search the space of such networks. However, for all but trivial problems, this space is very large. Hence, biases are required that guide the search. Here, we investigate solving a smaller related problem to establish such a bias. Networks are specified by trees containing operators that act on nodes (neurons) and edges (connections). We demonstrate the approach on a signal reproduction task that requires internal state. Performance on a small problem size was improved by solving a smaller problem first. By repeatedly applying the principle, versions of the problem were solved that were not solved by a direct approach.

1 Introduction

Recurrent Neural Networks (RNNs) can generate complex behavior [3]. A substantial problem with recurrent networks however is that they are difficult to design, due to the difficulty of predicting the behavior of elements that continually influence one another. Therefore, manual design of networks leads to a focus on designs that are relatively easy to construct and understand.

Instead of constructing networks by hand, another approach that has been used in the past is to *evolve* the architecture and/or weights of neural networks, e.g. [1]. However, since networks need to have a certain minimal complexity in order to be able to solve non-trivial problems, the search space for networks that potentially solve such problems is very large. Therefore, a useful *bias* is required that guides the search to promising areas of the space. A good source for such a bias is provided by solutions to related problems; if there is some similarity between two problems, then aspects of solutions to these problems may be reused. This approach may be seen as the evolutionary equivalent of *shaping* in machine learning, see e.g. [5]; related also is the idea of supplying the training set incrementally [6]. If these related problems speed up the search for the more difficult problem, then there may be a payoff in first solving such related problems as part of solving the actual problem. This is the case if the speedup resulting from the useful bias extracted from that problem compensates for the extra amount of effort spent on solving it. Since the search space of small networks is much smaller than that of large networks (the space grows exponentially with the number of elements), this approach may make it possible to solve large problems that are not solvable using plain search methods.

In this paper, we consider utilizing bias from small problems as an approach to finding neural networks that solve larger, related problems. An important consideration is the representation of networks, since this to a large extent determines the potential for reusing aspects that have been found useful in related problems. A specification in terms of a weight matrix does not appear suitable, since it is not clear how the weights of a small network can be used as a bias in finding a larger network that solves a related task, otherwise than simply reusing the network. Rather, what is required is a way to capture *construction principles* for the generation of successful networks. If networks can specified by a set of construction rules, then specifications that are found to be useful in a small problem may provide a good starting point for larger problems. Such a way of specifying networks in terms of construction rules is provided by Gruau's elegant *cellular encoding* scheme [9], variations of which have been studied by various researchers.

We describe a cellular encoding scheme that employs both node operators and edge operators. The expression process takes place on a two-dimensional plane. By allowing neurons to move in this space during the expression process, governed by the specification of the network, some potential for storing state during the ex-

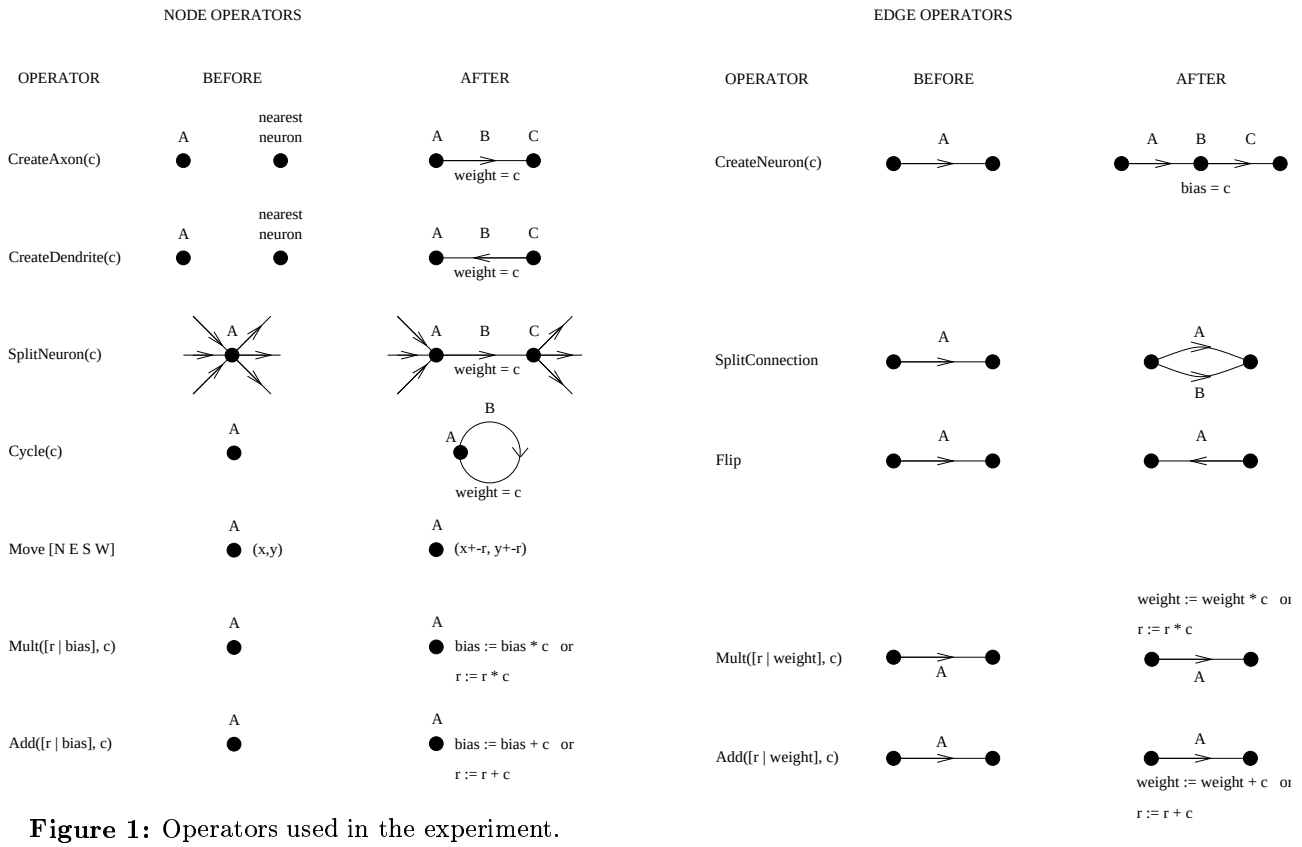

Figure 1: Operators used in the experiment.

pression process is provided. The motivation for this setup is that it may make the combination of partial network specifications easier.

The paper presents our first results following the above approach for the utilization of bias from earlier problems. The networks that are evolved are continuous time recurrent neural networks, which are tested on a sequence imitation problem. First, cellular encoding and its property of modularity are discussed. Then the task and the evolutionary method are described, followed by results and conclusions.

2 Cellular Encoding of Neural Networks

Cellular encoding was introduced by Gruau [9], and followed earlier generative approach to the specification of neural networks, e.g. [10]. A cellular encoding is a tree of operators. Gruau's description of cellular encoding employs node operators, i.e. all operators apply to cells and may result in one or more other cells, all of which eventually become neurons. Node-encoding has only weak control over the edges between the nodes, i.e. the connections between neurons. While this may be sufficient for the binary feed forward networks studied by Gruau, here specifications need to set the real valued weight of each connection, and thus appropriate control is required. Relevant to this purpose, the use of operators that operate on edges has been suggested [12]. Here, we employ both types of operators.

In the cellular encoding method used here, the expression of a tree into a network proceeds as follows. First, the operator at the root of the tree is applied to an initial neuron, functioning as a start symbol. This application may results in one or more new neurons and/or connections. Depending on the number of resulting neurons and connections produced by the operator, the operator has one or more child nodes. Each child node operator is applied to one of the products of the operator, and trees are expressed breadth-first.

The node operators used to modify neurons are the following, see figure 1. *CreateAxon* creates a connection that extends from the neuron, while *CreateDendrite* generates an input connection. Both operators link to the neuron whose spatial location is closest, excluding the neuron itself. Both also accept a parameter c, chosen randomly from the uniform distribution over $[-2, 2]$ at the time the operator is generated, which specifies the initial weight of the connection. The letters A, B and C in the figure indicate the elements to which children of the operator in the specification tree may be applied. *SplitNeuron* splits the neuron in two, assigns all input connection to the first and all output connections

to the second neuron, and creates a link from the first to the second neuron. *Cycle* creates a self-connection. *Move* moves the neuron in a specified direction (east, north, south or west), over a distance determined by an internal parameter of the neuron r. Furthermore, the arithmetic operators of multiplication and addition are available. They can be applied either to the parameter r of the neuron or to its bias weight. For multiplication, $c \in [-2, 2]$, while for addition $c \in [-1, 1]$.
The edge operators, applicable to connections between neurons, are the following, see figure 1. *CreateNeuron* inserts a neuron at the middle of the connection with an initial bias weight specified by the c parameter. *Split-Connection* splits the connection in two, and assigns half the original weight to each connection, so that the function of the network is disrupted as little as possible while introducing a potential for variation of the network's behavior. *Flip* reverses the direction of the connection. Finally, the *Multiply* and *Add* operators are available. These take either the internal parameter r or the weight of the connection as argument. Although r is not used by connections, it is inherited by connections from their parent neurons and passed on to child neurons, so that the distance over which neurons move can be varied and maintained by neurons of related ancestry.

3 Modularity of Cellular Encoding

A reason for using indirect encodings when evolving neural networks, as opposed to a direct encoding of the weights of a network, is that the information captured by a partial specification is more modular than a partial direct encoding. A module can be defined as a subset of elements whose interactions with other elements are limited, i.e. statistical dependencies with elements outside a module are reduced compared to a non-modular collection of elements.
In a direct encoding, the elements of variation are the weights themselves. While some weights have less effect on other weights than others, the representation of a weight matrix is unstructured, and hierarchical information about the influence of weights on sets of other weights is not directly available. In an indirect encoding such as cellular encoding on the other hand, which part of the network is affected by an operator is to a large extent determined by the position of the operator in the tree; whereas an operator at the top of the tree affects the complete expression process specified by the operators that follow it, the operators at the bottom of the tree can only make changes to the almost complete network that has been specified by operators at higher levels of the tree. Thus, specifications are hierarchically

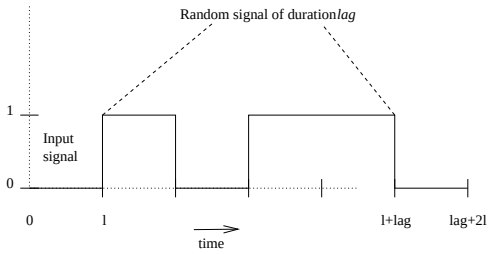

Figure 2: Illustration of the input signals used in the experiment.

organized, which results in a potential for the identification of useful modules. It may therefore be expected that cellular encodings are specifically suited to capturing information about the construction of successful networks.
Direct encodings of neural networks on the other hand are known to suffer from the *competing conventions* problem [1]. This problem directly results from the strong interactions between the elements; since changing one weight may affect the operation of many other elements in the network, it is difficult to combine sets of elements that have been formed independently. The use of crossover with direct encodings of neural networks may be problematic for this reason. In indirect encoding such as cellular encoding, the specifications are modular at least to some extent, and may therefore benefit from variation operators that combine independently adapted units, such as crossover.

4 Task Suite: Signal Reproduction

The question addressed here is whether it is possible to obtain a useful bias by first solving a similar but smaller problem. Thus, a task must be found of which instances of variable difficulty can be generated. Furthermore, since we investigate how recurrent neural networks can be evolved, the task should test the capacity specific to recurrent networks of constructing and using functional internal state [2], so that the potential of RNNs to generate and use internal state is exploited. A simple task that fits these criteria is signal reproduction, described as follows, see figure 2.
During an initial period $[0 \ldots l >$, a signal of duration $l = 1$ is presented. Since the period Δt after which the state of the network is updated equals 0.05, l corresponds to twenty updates of the network state. Next, during a time lag $[l \ldots l + lag >$, a distractive signal is administered. Then, during the final period $[l + lag \ldots lag + 2l >$, a base signal (zero) is presented, and the output of the network is monitored. The goal of the task is to reproduce the input signal during this

final period. Since the input signal is not available at this time, the network must consider the input signal, remember it in its internal, and subsequentially reproduce it. In the experiments presented here, the input signal is either zero or one. The duration of the distractive time lag lag is a multiple of l; it consists of randomly chosen zeros and ones, and may change only at multiples of l. The performance on the task is evaluated as the root mean square error (RMSE) over 10 sample points in the final period.

5 Evolutionary Method

Neural networks are encoded by trees containing the operators that have been described in section 2. The trees are of varying arity; its nodes may have one, two, or three children, depending on the operator they contain). Since such a tree completely specifies a network, we can apply evolutionary methods that operate on tree-shaped specifications, such as genetic programming, see e.g. [11]. Genetic programming employs two variation operators, called *crossover* and *mutation*. Crossover takes two trees, randomly selects a node from each, and exchanges the subtrees that have these nodes at their root. Mutation here visits all nodes in the tree and changes the operator they contain with probability $\frac{1}{n}$, where n is the number of nodes in the tree.
The evolutionary process starts by generating a random initial population of size $popsize$, containing trees with a number of operators chosen uniformly from $[10 \ldots 25]$. Next, at each generation, $popsize$ specifications are randomly selected from the population, and paired up to create a new generation by means of crossover, followed by mutation with $P = 0.1$. All individuals in the newly generated population are then evaluated on the task. Next, all individuals are compared to the current population. In addition to the evaluation on the task, this comparison involves two other objectives: the size of the specification tree, which should be minimized, and a diversity objective [4], which measures the average squared Hamming distance of the specification to the specifications in the current population.
To compare the multiple objectives of one individual with that of another, techniques from evolutionary multi-objective optimization may be used, see e.g. the overview by Fonseca and Fleming [8] for an introduction. Here, we follow Fonseca and Fleming [7] by counting the number $nrdom$ of individuals that dominate a given individual as a basis for selection. An individual A dominates another individual B if it performs at least as well on all objectives, and better in at least one objective. We sort the population according to $nrdom$,

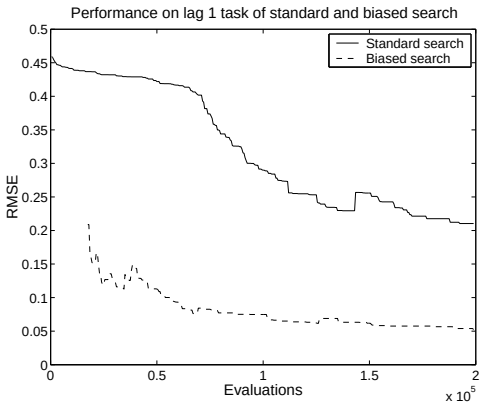

Figure 3: Root mean squared error on the lag 1 task of standard search and search biased by solutions to the lag 0 problem.

and select the top ranking $popsize$ individuals to yield the next generation.

6 Results

In the first experiment, the most basic case of solving a simpler problem first is investigated. The first problem in the sequence of signal reproduction problems is the lag 0 problem, where the input signal has to be reproduced directly after the first time step, i.e. there is no time lag in between the presentation of the input signal and the period during which the outputs are monitored. In the next, more difficult problem, a time lag of duration 1 is inserted between the input and output periods. During this time lag, the input signal randomly takes a value of either zero or one. Each network is tested on ten samples for both the *zero* and the *one* input signal, so that a network that uses the input during the time lag to determine its output is unlikely to be successful. For the increasing problem size experiment, we set up the simulation such that when a performance threshold is reached (RMSE 0.1), the problem size is increased from the lag 0 to the lag 1 problem. During a problem size increase, the best solution from the population is preserved, the rest of the population is discarded, and a new population is created that consists of random specifications (75%) and copies of the best solution to the lag 0 problem (25%). In the comparison experiment, the standard search to solving the lag 1 problem is taken, i.e. the lag 1 problem is used during the whole experiment. For both experiments, ten runs were performed.
Figure 3 shows the results for both experiments. The standard approach, solving the lag 1 problem, starts off with a slowly decreasing error. The performance

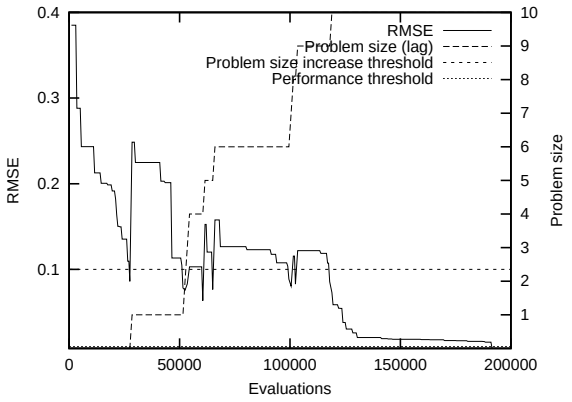

Figure 4: Example of a run in the experiment with problem size increasing from lag 0 to lag 10.

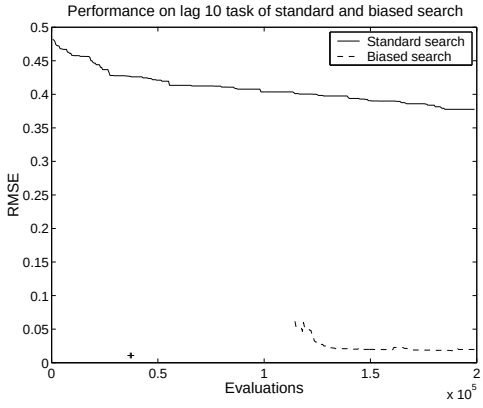

Figure 5: Root mean squared error on the lag 10 task of standard search and search biased by sequentially solving the lag 0 through lag 10 problems.

gradually improves, but does not get better than an RMSE of 0.2 on average within the duration of the experiment.

In the increasing problem size experiment, the lag 0 first has to be solved with an accuracy of RMSE 0.1. Different runs solve this problem at different points in time, of course, but all of the ten runs solved the problem within 39,000 evaluations. After generating a new initial population including copies of the solution to this problem, the lag 1 problem can be addressed. The bias conferred by this technique is clearly helpful. Not only does the biased search start off at a better performance (around 0.2, the final value reached by the standard approach), but the error continues to drop. This indicates that the lag 0 solutions with which the populations were seeded were not merely reasonable solutions to the lag 1 problem, but that aspects of these specifications could be fruitfully applied in the lag 1 problem. As the graph shows, the performance improvement due to the bias from the lag 0 problem more than compensates for the time spent on solving this problem first.

In the second experiment, the principle of solving a small problem and using the solution to solve a more difficult problem is applied repeatedly. After solving the lag 0 problem, the search continues to the lag 1, 2, 3...10 problems. When the lag 10 problem is reached, the problem is kept fixed, so that the search can continue to improve performance on the problem.

Figure 4 shows an example run of the repeatedly increased problem size experiment. After 28,500 evaluations, performance on the lag 0 problem reaches the 0.1 threshold. Consequently, the problem size increases to the lag 1 problem. This results in a drop in performance to an RMSE of around 0.25. Within another 25,000 evaluations however, the lag 1 problem is solved, and the problem size increases to 2. The process continues until problem size 10 is reached. The problem size then remains constant, and eventually the performance threshold of RMSE 0.01 is reached and the run terminates.

Figure 5 shows the performance on the lag 10 problem of the five (out of ten) runs that reached this problem size before 200,000 evaluations. One of these runs found a solution very early on (marked by the '+'); this solution was found during the search for the lag 0 problem, but since its performance was below RMSE 0.1 on the subsequent problems as well, the problem size increased at each of the ten following generations. Subsequently, the performance threshold of 0.01 was reached, and the run terminated. The dashed line shows the average performance of the other four runs. The final performance is even better than for the lag 1 experiment, which can be explained by the fact that only the five runs reaching problem size 10 are measured, while performance in the lag 1 experiment is averaged over all of the ten runs. Of the five runs that reached problem size 10, four reached the performance threshold of RMSE 0.01, and thus solved problem.

The comparison experiment faces the difficult task of discovering the dependency between the input signal at the first time step, the output signal at the final time step, and the resulting fitness. This relation is obfuscated by the ten time steps during which a random signal is presented, and is therefore very difficult to discover, as reflected in the average error, which remains above 0.35. All errors remained above 0.3, and thus none of the runs reached the performance threshold of 0.01.

The transfer shown here is admittedly rather direct, and the problem is somewhat basic compared to the complexity of the evolutionary machinery. The experiments should be seen as a first demonstration of the

principle of using small problems to establish useful bias for solving larger related problems. While both directly [14] and indirectly [13] encoded solutions to related problems have been used before, cellular encoding appears particularly suited for this purpose. Perhaps it may be possible to apply the potential of this approach to problems for which network architectures would otherwise be practically impossible to find.

7 Conclusions

Architectures and weights for recurrent neural networks may be found by searching the space of such networks. Since this space is very large, a bias is required to guide the search. In the current article, we investigate the use of smaller, related problems to establish this bias.

We described and used a variant of Gruau's *cellular encoding* that uses both node operators and edge operators. Thus, more control over connections between neurons is achieved compared to the original node encoding approach. The motivation for using an indirect encoding of the network is that the rules for the construction of networks found this way may be more suitable for capturing information about network construction from related problems than a direct encoding. The problem on which the approach was tested is sequence reproduction with a distractive time lag of variable length. First solving the zero time lag problem considerably improved performance on the lag 1 problem compared to starting on the lag 1 problem directly. Thus, a useful bias was gained from solving the small version of the problem first. The same technique was applied repeatedly, up to lags of 10 time steps. The repeated transfer of solutions from small problems to larger problems sizes allowed the evolutionary search process to find solutions to the lag 10 problem, while the comparison experiments that attempted to solve this problem directly failed to do so.

The results are an initial demonstration of the idea that the difficult problem of searching for recurrent neural network architectures and weights may be addressed by incrementally establishing useful biases that guide the search.

References

[1] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1993.

[2] P.B. Bakker and M.B. De Jong. The epsilon state count. In J.-A. Meyer, A. Berthoz, D. Floreano, H. Roitblat, and S.W. Wilson, editors, *From Animals to Animats 6: Proceedings of The Sixth International Conference on Simulation of Adaptive Behavior*, pages 51–60, Cambridge, MA, 2000. MIT Press.

[3] Randall D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, 1995.

[4] Edwin D. De Jong, Richard A. Watson, and Jordan B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), to appear*, 2001.

[5] Marco Dorigo and Marco Colombetti. *Robot Shaping*. MIT Press, Cambridge, MA, 1998.

[6] Jeffrey L. Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–9, July 1993.

[7] Carlos M. Fonseca and Peter J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA'93)*, pages 416–423, San Mateo, California, 1993. University of Illinois at Urbana-Champaign, Morgan Kauffman Publishers.

[8] Carlos M. Fonseca and Peter J. Fleming. An Overview of Evolutionary Algorithms in Multiobjective Optimization. *Evolutionary Computation*, 3(1):1–16, Spring 1995.

[9] Frederic Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, PhD Thesis, Ecole Normale Supérieure de Lyon, 1994. anonymous ftp: lip.ens-lyon.fr (140.77.1.11) directory pub/Rapports/PhD file PhD94-01-E.ps.Z (english) PhD94-01-F.ps.Z (french).

[10] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476, August 1990.

[11] J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[12] Sean Luke and Lee Spector. Evolving graphs and networks with edge encoding: Preliminary report. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 117–124, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[13] Eric Mjolsness, David H. Sharp, and Bradley K. Alpert. Scaling, machine learning, and genetic neural nets. *Advances in Applied Mathematics*, 10, 1989.

[14] B. M. Yamauchi and R. D. Beer. Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, 2(3):219–246, 1994.