Genomic Computing Networks Learn Complex POMDPs

David Montana, Eric VanWyk, Marshall Brinn, Joshua Montana and Stephen Milligan
BBN Technologies
Cambridge, MA, USA
{dmontana,mbrinn,milligan}@bbn.com, eric.vanwyk@students.olin.edu, jdmontana@gmail.com

ABSTRACT

A genomic computing network is a variant of a neural network for which a genome encodes all aspects, both structural and functional, of the network. The genome is evolved by a genetic algorithm to fit particular tasks and environments. The genome has three portions: one for specifying links and their initial weights, a second for specifying how a node updates its internal state, and a third for specifying how a node updates the weights on its links. Preliminary experiments demonstrate that genomic computing networks can use node internal state to solve POMDPs more complex than those solved previously using neural networks.

Categories and Subject Descriptors: I.2.6
General Terms: Algorithms
Keywords: evolutionary neural networks, POMDP

1. INTRODUCTION

Current neural networks can perform certain functions, but typically only as well or incrementally better than statistical or other traditional techniques. Genomic computing networks provide the following additional functionality that expands the capabilities of neural networks:

- The nodes do not implement a simple transfer function, such as a sigmoid, but rather have complex internal state and operations matched to their function.
- The nodes have weight update rules (i.e. learning rules), matched to their particular role/function.
- Nodes can form large, heterogeneous structures with varying patterns of connectivity and functionality.
- The structure and function of the nodes and their networks are designed by evolution.

Genomic computing networks embody an ambitious vision that cannot be achieved all at once. We describe some initial steps towards validating the approach.

There has been a large amount of work in the area of evolutionary algorithms applied to neural networks. One aspect of our work that distinguished it from most of the rest is that it is simultaneously evolving not only the architecture and initial weights but also the learning rules and internal node operation. Hussain [3] has the same broad goals, but uses a very different implementation involving grammars and genetic programming.

Most of the problems described in Section 3 involve reinforcement learning with memory, also referred to as partially observable Markov decision processes (POMDPs) [1]. These are problems where knowledge of the current state is not sufficient, and the decision process needs to remember the past. There have been a range of neural network approaches applied to such problems, including pure recurrent networks (e.g., [4]) and explicit memory cells [2]. Our more general approach is to provide general-purpose internal state for the nodes and to allow evolution to determine how to use it.

Copyright is held by the author/owner(s).
GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
ACM 1-59593-186-4/06/0007.

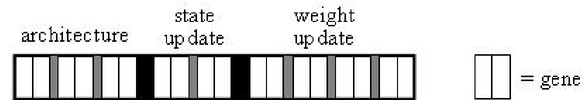

Figure 1: Structure of the genome

condition 1			condition 2			state to modify	coefficient	exponents				
state	min	max	state	min	max							
3	-1.4	0.8	1	0.4	4.2	2	-0.7	0	0	1	0	-1

Figure 2: This example state update gene specifies to add -0.7*state2/state4 to state2 if -1.4 < state3 < 0.8 and 0.4 < state1 < 4.2.

2. GENOMIC COMPUTING NETWORKS

Node and Network Basics - The basic network structure of a genomic computing network is like that of standard neural networks. A big difference is that genomic computing networks use internal state. The different types of state are:

Location states identify the location of the node with respect to the geometry and topology of the network.

The *input state* is the weighted sum of the inputs from the incoming connections to a node.

Free internal states are set according to the state update rules in the genome.

The *output state* is a special free internal state.

Weights can be viewed as a special type of node state.

The *feedback states* provide data that can be used by the weight update rules, including information on how downstream nodes change their weights.

The Genome - The details of not just the structure of a genomic computing network but also its operation are encoded in a genome. As illustrated in Figure 1, the genome has three sections. Each section controls a different aspect of the network and its functionality: one for the connectivity and initial weights (architecture), one for the state update procedure, and one for the weights update procedure. Each is further subdivided into a variable number of individual genes. Every gene has two parts, a regulator and an action. The regulator consists of a set of conditions that must all be satisfied before a node can execute the action. Heterogeneity arises from only certain nodes expressing each gene (i.e., executing the action associated with that gene).

The genes have different forms in each of the three sections

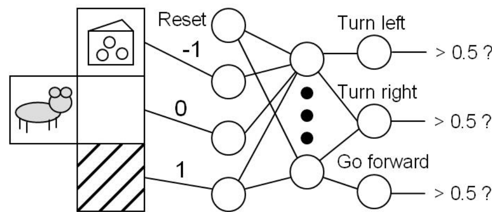

Figure 3: The control logic

of the genome, and here we focus just on the state update genes. The regulator portion consists of an arbitrary number (possibly zero) of conditions. Each condition specifies three values: which state to test and its minimum and maximum values. The action portion tells which state to modify and specifies the function that computes the quantity by which to increment/decrement the state's value. This function is a monomial in the input state and free internal states, with the coefficient and exponents given in the genome. When multiple active genes are associated with the same state, the monomials combine into a polynomial. Figure 2 shows an example.

The Genetic Algorithm - The base genetic algorithm code is a customized version of version 13 of ECJ that uses a master-slave mechanism for distributed processing. The genetic operators are each focused on one of the three components of the genome. For the state update genes, the operators are AddStateGene, DeleteStateGene, ChangeStateCoeffs, ChangeStateExps, ChangeStateConds, AddStateConds and CrossOverStateGenes.

3. EXPERIMENTS

We use two types of POMDP environments. One requires classification decisions based on sequences of ones and zeroes. The second is a simple virtual world involving a mouse, some cheese (its goal state), and mazes.

As shown in Figure 3, the mouse's "brain" contains control logic, implemented as a genomic computing network, that receives sensory inputs and produces decisions for how to move. Three sensory inputs tell the contents of the three squares it senses. The value of these inputs is 1 if the contents is a wall, 0 if the square is empty, and -1 if the square is the goal. A fourth input is 1 when a reset occurs (i.e., when the mouse is placed back at the beginning of the maze) and 0 otherwise.

The short summary of the experiments is that genomic computing networks were able to efficiently find solutions to each of the following test problems.

Majority-Ones Experiment - For an odd number, N, provide a sequence of N zeroes and ones at the single input node of the network. At the end, the output should be high if there were more ones than zeroes in the sequence and low if the opposite was the case.

N-In-A-Row Experiment - At the end of a sequence, the output should be high if at any point there were N consecutive 1's and low otherwise.

T-Maze-With-Counting Experiment - The T-maze problem has been previously used by multiple researchers to investigate reinforcement learning with memory [1]. A signal at the beginning of a long corridor tells an agent what direction to turn at the end of the corridor. We have added the extra challenge of counting N steps beyond the end of the corridor before turning. As shown in Figure 4, we formulate this as a mouse-and-maze problem with the signal being the lack of a wall on one side at the beginning of the hallway.

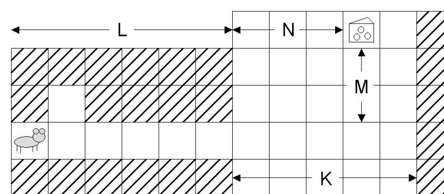

Figure 4: The T-maze-with-counting maze

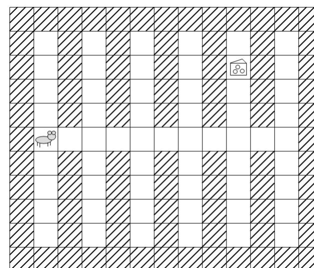
Figure 5: The many-branch maze

Many-Branch Experiment - This problem requires a simple maze search strategy combined with the ability to count steps and act accordingly. As shown in Figure 5, the maze consists of a main corridor with 10 different passageways branching off of it. The cheese can be in any of the 10 side passages but is always 3 steps in. An efficient search should not continue to the end of each side passage but rather turn around after 2 steps if the cheese is not there.

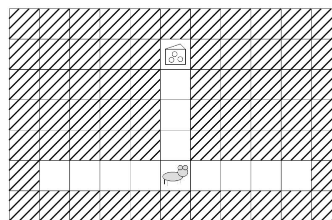
Figure 6: The 3-branch-exploration maze

3-Branch-Exploration Experiment - This problem differs in that it includes an explicit exploration phase during which the mouse gathers information about the maze. The mouse starts in the spot where three passageways diverge, as shown in Figure 6. The cheese is at the end of one of these passageways. The mouse is allowed three short exploration runs followed by an execution run, and is evaluated only on its performance during the execution run.

4. REFERENCES

[1] Gomez, F. and J. Schmidhuber: 2005, 'Co-Evolving Recurrent Neurons Learn Deep Memory POMDPs'. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO).*

[2] Hochreiter, S. and J. Schmidhuber: 1997, 'Long Short-Term Memory'. *Neural Computation* **9**(8), 1735–1780.

[3] Hussain, T.: 2003, 'Attribute Grammar Encoding of the Structure and Behaviour of Artificial Neural Networks'. Ph.D. thesis, Queen's University at Kingston.

[4] Stanley, K. and R. Miikkulainen: 2002, 'Evolving Neural Networks Through Augmenting Topologies'. *Evolutionary Computation* **10**(2), 99–127.