# Evolution of Internal Dynamics for Neural Network Nodes

**David Montana** (dmontana@bbn.com, Phone: 617-873-2719, Fax: 617-873-4086)
**Eric VanWyk** (vanweric@gmail.com)
**Marshall Brinn** (mbrinn@bbn.com)
**Joshua Montana** (jdmontana@gmail.com)
**Stephen Milligan** (milligan@bbn.com)
BBN Technologies
10 Moulton Street, Cambridge, MA 02138

**Abstract**

Most artificial neural networks have nodes that apply a simple static transfer function, such as a sigmoid or gaussian, to their accumulated inputs. This contrasts with biological neurons, whose transfer functions are dynamic and driven by a rich internal structure. Our artificial neural network approach, which we call *state-enhanced neural networks*, uses nodes with dynamic transfer functions based on n-dimensional real-valued internal state. This internal state provides the nodes with memory of past inputs and computations. The state update rules, which determine the internal dynamics of a node, are optimized by an evolutionary algorithm to fit a particular task and environment. We demonstrate the effectiveness of the approach in comparison to certain types of recurrent neural networks using a suite of partially observable Markov decision processes (POMDPs) as test problems. These problems involve both sequence detection and simulated mice in mazes, and include four advanced benchmarks proposed by other researchers.

**Keywords -** state-enhanced neural networks, neuroevolution, POMDP, dynamic neuron model

## 1 Introduction

Biological neurons have a complex and dynamic internal state, which includes the concentrations of proteins and other chemicals and the cell's physical structure. This state is constantly changing in response to not only the external stimuli the cell receives but also the neuron's internal processes driven by instructions from the genes. All the behavior of a neuron is the result of this complex interaction between genome, internal state, and external stimuli. This behavior includes development (division, specialization, etc.) and learning, but most importantly for this paper the computational process. The computational outputs from a biological neuron are a function of its internal state as well as its inputs, and can therefore reflect the history of all its inputs. In other words, a neuron has a dynamic response.

In contrast, the nodes in most artificial neural networks have no internal state. Instead, the computational output of each node is calculated as a static map from the input, where most commonly this transfer function is a sigmoid. Therefore, these nodes lack the richness of computational structure of biological neurons.

This paper investigates the use of a dynamic transfer function based on internal state in the nodes of artificial neural networks. As for biological neurons, the internal dynamics of the node (as specified by a set of state update rules) is determined by a (artificial) genome. Evolution acting on the genome optimizes the node internal dynamics for the particular role the node plays and the task that the neural network needs to perform.

Node internal state provides memory for the computational process. This memory can be either long-term or short-term, with the distinction being that the former is retained beyond the scope of a particular task to be performed and the latter is not. In this paper, we restrict our attention to short-term memory and its use to retain past observations and decisions to complete a task. For example, to perform a given task, an agent might need to remember that ten seconds ago a light blinked on its left side or that a few steps earlier it made a right turn. Because some tasks require maintaining this memory for relatively long periods of time, Hochreiter and Schmidhuber (1997) use the term *long short-term memory* to describe it; alternatively, Gomez and Schmidhuber (2005) refer to it as *deep memory*. The experiments of Section 4 investigate a variety of problems requiring deep memory and demonstrate the ability of state-enhanced neural networks to learn to perform such tasks.

[Note that the most common approach to short-term memory in neural networks is the purely manual one: create by hand some external structure for accumulating past observations and feed them into the neural network as inputs. For example, for time series prediction, there is usually some buffer external to the neural network that saves the last N values of the times series, which are provided as inputs to the neural network. However, for less structured problems it is not clear what observations to save, plus it is desirable for a learning algorithm to not require external assistance.]

Section 2 discusses related prior work. Section 2.1 provides a general overview of neuroevolution, i.e. evolutionary algorithms applied to the design of neural networks. Section 2.2 describes research on gene regulatory networks (GRNs). Like our work, the GRN work is concerned with interactions between an evolved genome and node internal state. The difference is that the GRN approaches focus on development while ours focuses on the computational process. (Note that these two approaches are potentially complementary as described in Montana et al. (2006).) Section 2.3 discusses other neural networks that use nodes with dynamic transfer functions. Unlike our approach, which uses a genetic algorithm to optimize the dynamical behavior of the nodes, the node internal dynamics for these other types of networks are defined by hand. Section 2.4 discusses other approaches to the solution of POMDPs, particularly those requiring deep memory. To date, some of the most effective general approaches to short-term memory internal to neural networks use recurrent networks trained by evolutionary algorithms. These include ESP by Gomez and Schmidhuber (2005), CTRNN training by Blynel and Floreano (2003), and Evolino by Schmidhuber et al. (2007). We compare the performance of our approach with these techniques on problems similar to the benchmarks that they utilized.

Section 3 discusses state-enhanced neural networks. Section 3.1 details the underlying model we use for internal state and the state update rules that determine its (nonlinear) dynamic behavior. These state update rules are encoded in a genome that is described in Section 3.2 and is similar to the genomes used in GRN approaches. Each gene specifies a change to the value of a single state variable as a function of (i) the values of other state variables and (ii) the aggregate input to the node from other nodes. The genome also contains the values for all the connection weights. Section 3.3 describes the genetic algorithm that optimizes the genome, and hence the state update rules and connection weights, for different tasks.

Section 4 describes experiments that evaluate the effectivness of our approach. We have defined a set of different test problems, all of which are *partially observable Markov decision processes* or POMDPs. POMDPs are a class of decision problems for which current decisions cannot be based on just current observations but also must consider past observations and decisions. We wanted to use a variety of different test problems to demonstrate that our technique is generally applicable and not just suited to a single problem or domain. For memory-based decision problems, there is no standard suite of tests analogous to the UCI Machine Learning Repository for pattern classification problems administered by Asuncion and Newman (2007). Therefore, we have defined our own test suite containing both sequence detection and mouse-in-maze problems. Some of the problems were selected because they are the same or similar to benchmarks used to evaluate competing techniques and hence provide a means for comparing with other approaches. The other problems were selected to showcase the capabilities of our approach in ways previously not demonstrated for other techniques. All the problems can be solved using state-enhanced neural networks. Hence, the experiments show that our technique provides capabilities to solve deep-memory POMDPs at least comparable to those demonstrated for the best alternative neural-network approaches.

## 2 Previous Work

We now provide a summary of some of the previous work most closely related to ours.

### 2.1 Neuroevolution: A Brief Overview

There has been a large amount of work in the area of *neuroevolution*, i.e. evolutionary algorithms applied to neural networks, too much to discuss all of it in detail here. A good summary of the field, although a bit out-of-date at this time, is given by Yao (1999). Another good summary that focuses on the more recent work is by Floreano et al. (2008). We refer the reader interested in the entire field to these papers. In this paper, we provide a brief overview before focusing in greater detail on that work most closely related to ours.

The two characteristics that are most important in distinguishing between the wide range of different approaches to neuroevolution are (i) what aspect(s) of the neural network is(are) being optimized, and (ii) how different solutions are represented as genomes. The different aspects to optimize include

- Connection weights: In most neural networks, the aggregate input to a node is the weighted sum of all its inputs. The selection of values for the connection weights is one way to vary the network's behavior. Evolutionary optimization is one approach that has long been used for weight selection starting with Montana and Davis (1989) and Whitley (1989). As described in Section 3.2, the genome used in this paper has a section for encoding the weights.

- Architecture: Another important factor determining network functionality is how the nodes are connected. Evolutionary algorithms can be used to optimize the architecture for particular tasks. Originally, Harp et al. (1989) used them for optimizing just feedforward network architectures, but later others such as Stanley and Miikkulainen (2002) used them for more complex and powerful architectures, such as recurrent networks.

- Learning rule: Evolutionary algorithms can optimize the method for updating weights and/or connections. For example, Chalmers (1990) and Baxter (1992) demonstrated genetic algorithms finding weight update rules that were alternatives to standard rules, such as Hebbian learning and backpropagation.

- Node behavior: Evolutionary algorithms can optimize the functionality of the nodes. Usually, this involves just picking from a menu of different tranfer functions, e.g. sigmoid or threshold or Gaussian, as done by Hussain (2004). Optimizing more complex behavior is the primary focus of this paper.

It is now the norm to optimize multiple aspects of the neural network with a single genetic algorithm. For example, Stanley and Miikkulainen (2002) optimize the architecture and weights of a recurrent network simultaneously. Abraham (2004) jointly evolves the architecture, weights, and learning rule. Hussain (2004) optimizes all four above-mentioned aspects at once.

With regards to the genome representation, an important distinction, although one that is not always clear-cut, is that between direct encodings and indirect encodings. A direct encoding is one that explicitly encodes the properties of the neural network that it is optimizing. Indirect encoding uses a representation where the genome contains instructions that need to be executed to determine the structure and/or function of the network. There is a wide variety of approaches to indirect encoding. Some of the original work on indirect encoding was done by Kitano (1990), who uses a genome that contains matrix rewriting rules, in what is essentially a grammar-based encoding, to specify the network structure. The cellular encoding approach invented by Gruau (1995) uses an instruction tree created by genetic programming to guide how the cells divide and connect. Cellular encoding introduced the concept of the *developmental* approach, where the neural network starts as a single node/cell, known as an *embryo*, and grows through a sequence of steps to its full structure. We now discuss a class of developmental approaches closely related to our work.

### 2.2 Gene Regulatory Networks (GRNs) for Neuroevolution

The process of growing virtual cellular structures is referred to as *artificial embryogeny* (or alternatively, *computational embryogeny* or *morphogenesis*), and it is a topic of significant recent interest including overviews by Stanley and Miikkulainen (2003) and Kumar and Bentley (2003). Closely related to our work is a class of such algorithms referred to by Stanley and Miikkulainen (2003) as *cell chemistry approaches*. These are based on abstractions of the cellular processes inside a biological neuron, where the levels of different biochemicals in a real neuron are equivalent to the values of different (binary or real-valued) state variables in an artificial neuron. The genes in the genome specify how to change these states/levels during each time step. The focus of these approaches tends to be on the development process, and hence network architecture, not the computation process (as in our work).

An early example of the cell chemistry approach is the work done by Nolfi et al. (1994), where the state of a neuron is the cell's axons and their lengths. The genes specify the branching and positional properties of the neurons, and hence how to grow the connections between the neurons. This was later enhanced by Cangelosi et al. (1994) to include cell division and cell migration. Astor and Adami (2000) used a low-level and detailed model of a cell with the artificial chemicals diffusing according to something resembling physical laws.

Most of the work in this area uses artificial analogs of gene regulatory networks

(GRN). In biology, a GRN is a feedback loop involving genes and proteins. The genes code for the creation of proteins (and indirectly for other types of biochemicals), while the presence of certain proteins turns the genes on or off. Each gene has two parts, a regulatory portion and a coding portion. The regulatory portion determines whether the gene is expressed (i.e., is turned on), while the coding portion creates a particular protein when the gene is expressed.

Dellaert and Beer (1996) did some of the first work using an abstracted GRN, applying it to the evolution of dynamical neural networks. All the actions of the neurons, including communication with other cells, are governed by the presence or absence of certain *proteins* in the cell, where each protein is actually just a binary state variable. A gene is a binary expression that dictates to turn a particular state/protein on or off based on the values of other states.

The work of Jakobi (1995) is similar to that of Dellaert and Beer (1996) in its use of GRNs to control the actions of neurons as they form a network, in this case a recurrent network. The big difference is that proteins are, like their biological counterparts, strings of nucleic acids, in this case represented as characters. The regulatory portion of a gene is based on matching substrings of these proteins.

Bongard (2002) used a GRN encoding as a means to evolve an agent's brain and body simultaneously, evolving agents that could locomote in a simulated world. The states of the GRN are real-valued, which means that the model tracks the concentration of different "proteins" and not just their presence or absence. Like Bongard (2002), Eggenberger-Hotz et al. (2002) utilized real-valued concentration levels for the proteins. There are complex equations governing how these levels change and how the interactions between genes and proteins occur, with each gene containing a set of parameters that controls the process.

## 2.3 Dynamic Neuron Models

We now examine other work that uses neurons with internal state. In this case, the output of a neuron is not simply a function of its current aggregate input. Floreano et al. (2008) refer to such approaches as *dynamic neuron models*. Generally, these neuron models use a single internal state variable with predefined dynamics. (In contrast, our approach uses multidimensional internal state with the dynamics optimized to the task.)

One such neuron model is the *leaky integrator*, which is described by Haykin (1999) and which is essentially a one-pole low-pass filter. It performs integration, summing its inputs over time, but old values decay so that the newer inputs are weighted more. One use for such neurons is in continuous-time recurrent neural networks (CTRNN), as described by Pearlmutter (1995) and Funahashi and Nakamura (1993). Such networks are good for estimation of time-variant systems. In addition, Blynel and Floreano (2003) use CTRNN for the T-maze problem, which as discussed in Section 4.4 is a problem to which we apply our state-enhanced neural networks.

Another set of dynamic neuron models are those involving spiking neurons (Gerstner and Kistler (2002)). There are a large variety of such models, and Izhikevich (2004) enumerates them systematically. One common spiking neuron model is the integrate-and-fire neuron, which integrates all its inputs over time until it reaches a threshold, at which point it outputs a pulse and resets its internal state. The dynamic nature of such neurons potentially allows networks with such neurons to solve problems that other networks cannot, as shown by Floreano and Mattiussi (2001). One alternative spiking neuron model is the Z-model, described by Ichinose et al. (1993).

Yet another set of dynamic neuron models are those referred to as *chaotic* neurons. The state update equation for a neuron is nonlinear and leads to chaotic behavior. Such neurons have been used for associative memory, e.g. by Aihara et al. (1990) and Osana et al. (1996).

In some cases, evolutionary algorithms are used to optimize the architecture and/or weights of networks with dynamic neuron models. Examples of such work include that by Yamauchi and Beer (1994) and Blynel and Floreano (2003) on evolutionary optimization of CTRNNs and research by Floreano and Mattiussi (2001) on evolutionary optimization of spiking neural networks. However, unlike in our work, these are not optimizing the neuron model, i.e. the dynamic behavior of the nodes, but rather the structure of the network containing the nodes.

### 2.4 POMDPs and Recurrent Networks

The experiments described in Section 4 involve *partially observable Markov decision processes* (POMDPs). These are a class of problems where knowledge of the current state is not sufficient to solve the problem, and the decision process needs to remember the past. (Partial observability means that certain aspects of the current state affecting the outcome of the current decision are not observed by the decision maker.) Aberdeen (2003) provides a good general overview of different solution techniques, most of which fall under the general category of reinforcement learning. We are particularly interested in the model-free techniques, since they learn to solve the problem with no a priori knowledge, as we would like to do. Included among these are a variety based on dynamic programming, such as Q-learning (Watkins and Dayan (1992)), and others based on hidden Markov models, such as that proposed by McCallum (1993).

The solution techniques of most interest to us are those based on neural networks. The majority of these use recurrent networks, which can utilize their feedback connections to recirculate values and hence provide simple short-term memory. Indeed, Funahashi and Nakamura (1993) have shown that continuous-time recurrent neural networks (CTRNNs) are universal dynamic approximators, although training methods to realize the theoretical capabilities of CTRNNs have been difficult to find. A particularly successful approach to training recurrent network uses evolution to set the weights and also potentially determine the topology, with examples being the NEAT method of Stanley and Miikkulainen (2002), ESP method of Gomez and Miikkulainen (1999), and evolutionary training of CTRNNs of Blynel and Floreano (2003). Gomez and Schmidhuber (2005) have demonstrated the use of recurrent networks evolved by ESP to solve *deep-memory POMDPs*, i.e. POMDPs that require the decision maker to remember far into the past. The problem they solved is the T-maze signal problem with a long corridor described in Section 4.4. Additionally, Blynel and Floreano (2003) used CTRNNs to solve the T-maze exploration problem described in Section 4.7.

An alternative neural-network approach is to use hand-designed neural components to provide memory. An example is the use of what Hochreiter and Schmidhuber (1997) refer to as *long short-term memory cells* as structures separate from the general-purpose nodes. Gers and Schmidhuber (2001) and Bakker et al. (2003) show that this technique also can solve deep-memory POMDPs such as the T-maze problem and sequence recognition problems. This original work on long short-term memory used gradient-based methods for training the specialized recurrent networks. In contrast, Schmidhuber et al. (2007) and Schmidhuber et al. (2005) developed an approach called Evolino using evolutionary algorithms, among other techniques, for learning the connection weights of these networks. This has improved the performance of the net-

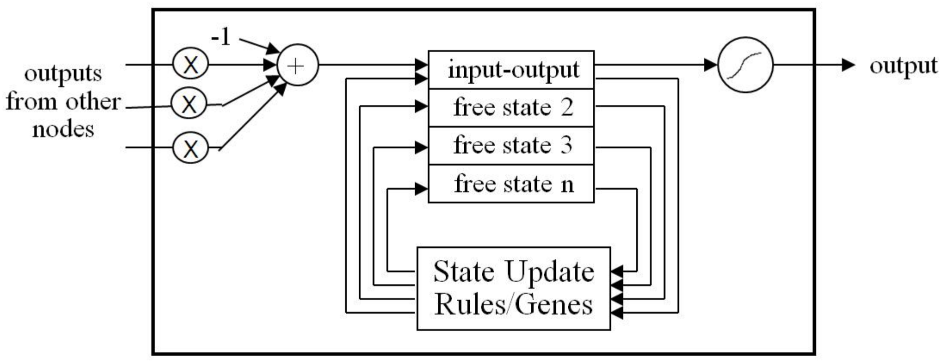


Figure 1: The internal operation of a node

works, and the technique has proven capable of solving an impressive variety of problems including the $a^n b^n c^n$ grammar problem described in Section 4.3. We compare our work against Evolino and other competitors to the extent possible and practical.

## 3 State-Enhanced Neural Networks

We now describe the various aspects of our approach, which we refer to as *state-enhanced neural networks*.

### 3.1 The Node and Network Models

The basic network structure we use is like that of standard neural networks. It contains a set of nodes connected with directional links, with each link having an associated weight. Each node is either an input node, an output node, or a hidden node. An input node has no incoming links and has its output value set externally, providing a way for external stimuli to enter the network. Each non-input (hidden or output) node computes a weighted sum of the output values on its incoming links and uses this to compute its output. An output node differs from a hidden node only in being designated as where to read output from the network.

The difference between our networks and standard neural networks is what happens internal to a non-input node. For each time step, the node's internal state is updated according to the state update rules encoded in the genome, and the node's output is computed based on the internal state. The internal state variables are real-valued, and they represent the current state of the node's computation. There is a specified number, $n$, of such state variables (which we henceforth refer to as just *states*), and hence the computational state of a node is essentially an $n$-dimensional real-valued vector. All but one of these internal states is hidden, i.e not accessible outside the node. Figure 1 illustrates the internal structure supporting a node's computation.

The *input-output state* is a special internal state, the only one that is externally accessible. At the beginning of each time step, its value is set to one less than the weighted sum of the inputs from incoming connections to the node. [Subtracting one from the weighted sum was found to improve performance in some preliminary experiments we used to refine the technique. In theory, this adjustment is not necessary, since the internal dynamics of the node can be arbitrary and hence can compensate for this difference, as discussed in Section 3.2. In practice, subtracting one makes it easier to find a solution, similar to the biases in the genetic operators discussed in Section 3.3. We

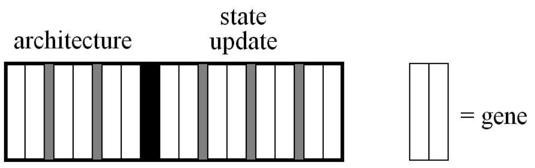


Figure 2: The genome contains two sections, a fixed-sized section specifying the connection weights and a variable-sized section specifying the state update rules.

hypothesize this is because it allows the value derived from a single input to have either a positive or negative sign despite the input always being positive.] At the end of each time step, its value is transformed by a sigmoid and used in computing the value for the node's outgoing connections. Note that if the state update genes specify to not change the value of the input-output state, then the computational behavior of the node is like that of a traditional neural network.

The internal states other than the input-output state are referred to as *free* internal states. They provide the memory for the node's computation process.

There are two nested cycles involved in the dynamic updating of the network. The outer cycle consists of the values of the input nodes changing to reflect the changes in the external stimuli of the network. During the inner cycle, the non-input nodes iterate through the following steps:

1. Set the value for the input-output state to be the weighted sum minus one of the node's inputs.

2. Compute updated values for the internal states, including the input-output state, by following the rules in the state update genes.

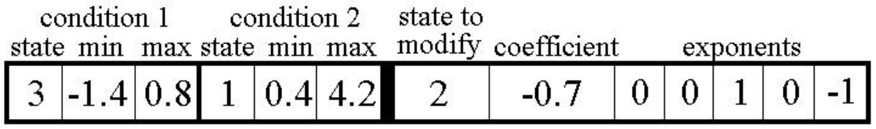
The inner cycle is repeated some constant $N$ times for each iteration of the outer cycle, i.e. each change in the external stimuli. The value of $N$ is currently a hand-selected parameter, determined partly by the structure of the network. For a feedforward network, such as all those used in the experiments described below, $N$ should be large enough to allow the values to propagate forward to the output nodes. In recurrent network, $N$ should be large enough to allow the network to settle. Note that there is no guarantee that a particular recurrent network will settle, but a well-chosen evaluation function will ensure that those that do not settle will have low fitness and will be eliminated by the genetic algorithm.

### 3.2 The Genome

The genome we define has two sections, one specifying the weights for the connections and the other specifying the state update rules, i.e. internal dynamics. For the purposes of the work in this paper, we constrain the network to have a fixed feedforward architecture. So, there are a fixed number of genes in the first section of the genome, where a gene is a real value specifying the weight for a particular connection.

The section of the genome defining the state update rules has a variable number of genes. Each state update 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 (or, in more biological terms, before the gene can be expressed). Note that this form for a gene is similar to those used for GRN approaches.

The regulator portion can have an arbitrary number (possibly zero) of conditions. Each condition specifies three values: (i) which state to test, (ii) the minimum value



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

Figure 3: 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.

this state can have, and (iii) the maximum value of this state. For the example gene in Figure 3, there are two conditions. The first condition checks whether state 3 is between -1.4 and 0.8, while the second tests whether state 1 is between 0.4 and 4.2.

The action portion of a state update gene tells which state to modify and specifies the function that computes the quantity by which to increment or decrement the state's value. The function is specified as a monomial in the internal states, with the coefficient and exponents given in the genome. The exponents can be either positive or negative. For the example gene in Figure 3, the state to modify is state 2, the coefficient of the monomial is specified to be -0.7, and the non-zero exponents are 1 for state 2 and -1 for state 4. Therefore, when the gene is active, -0.7*state2/state4 is one term added to the old value of state 2 to compute the new value of state 2. If there are additional active genes (i.e., genes whose conditions are met) specifying how to change state 2, the different contributions are all added to the value of state 2.

This method of representing state update rules allows specifying the change in the state per update cycle as an arbitrary polynomial function of the current state. For a single state variable state $i$, there will be some number $n_i$ of active genes specifying a quantity to add to the value of state $i$ each update cycle. For each such gene, the quantity is a monomial involving all the different state variables. Summing all the quantities from these genes yields a polynomial. Since there can be an arbitrary number of such genes each specifying an arbitrary monomial, it is possible to represent an arbitrary polynomial. While a formal analysis of universality is beyond the scope of this paper, Taylor series analysis shows that any well-behaved function can be approximated arbitrarily closely by a polynomial, at least in a certain neighborhood. So, this genome provides great generality in its ability to represent arbitrary state update rules, and hence arbitrary internal node dynamics.

The regulator portion of the genes provides the ability to change the internal dynamics based on the state. This would be most useful when there is a second type of state variable that remains fixed during the computation process and that indicates the role of the node. In this scenario, nodes can differentiate based on role, with different nodes using different dynamics all based on the same genome. Regrettably, we have not yet investigated experimentally inclusion of one or more state variables that just specify role, so we do not know the effectiveness of the regulator portion for node differentiation.

While the genome in theory allows representation of nearly arbitrary intenal node dynamics, it is currently not possible in practice to search this space fully. In the next subsection, we discuss how the genetic algorithm biases the search to a portion of the search space where a good solution is more likely. This does raise the question of whether the genome itself should be more restrictive in the dynamics it can represent in order to limit the size of the search space.

### 3.3 The Genetic Algorithm

We assume that the reader is familiar with a standard genetic algorithm and focus on those aspects of our genetic algorithm that are noteworthy.

The base genetic algorithm code is a customized version of release 13 of ECJ, which was originally developed by Luke (2002).

We use a steady-state replacement strategy, generating and replacing a single individual at a time rather than the whole population. For all the experiments, parent selection was performed using tournament selection with a tournament size of seven. Selecting which individual to remove from the population used a tournament of size ten with the worst of the candidates selected.

We varied the population size for each experiment, with a larger population used for more difficult searches. Since it is steady-state, there are no generations. Therefore, the amount of work is measured by the number of evaluations. For reporting purposes, it is sometimes convenient to convert to *pseudo-generations*, the number of evaluations divided by the population size.

Terminating the genetic algorithm was usually done by the human experimenter after it appeared that the genetic algorithm was converged and not improving its best solution. The time for the run is reported as the first pseudo-generation at which this best solution, or an equivalent, was found.

#### 3.3.1 The Genetic Operators

For the experiments discussed below, we used the following set of genetic operators. The last of these is a crossover (i.e. uses two parents) and the rest are mutations (i.e. use one parent).

- ChangeWeights mutates the weight value of each connection gene with probability 0.8. The mutation adds a random number between -1 and 1 to the weight with probability 0.2 and multiplies it by a random number between -2 and 2 with probability 0.8.

- AddStateGene adds a new state update gene and initializes it as described below in the discussion of the initialization procedure.

- DeleteStateGene randomly selects a state update gene to delete.

- ChangeStateCoeffs selects a new random value for the coefficient of each state update gene with probability 0.4. The new value is obtained by adding a random value between -4 and 4 to the current value.

- ChangeStateExps selects a new random value for one exponent of each state update gene with probability 0.6. The new value is obtained by setting this exponent to a random integer between -1 and 1 with probability 0.7, or by adding a random integer between -2 and +2 to the exponent with probability 0.3.

- ChangeStateConds, for each state update gene with probability 0.7, changes the range for each condition with probability 0.7. The range is always doubled in size with equal probability of either extending the bottom or top of the range.

- AddStateConds adds to each state update gene with probability 0.7 a new randomly generated condition.

- CrossOverStateGenes creates a child that has the same architecture/weight genes as the first parent and includes each state update gene from each parent with probability 0.5.

Note that there is a strong bias in the genetic operators for certain parts of the genome space over others. Most obviously, but not uniquely, ChangeStateExps is biased strongly towards exponents whose values are -1, 0, or 1. (As we discuss in the next subsection, the initialization procedure also has a similar bias, particularly towards exponents being 0.) In practice, this bias helps speed search.

Further note that while these operators were sufficient for the experiments described below, there should be much room for improvement, which will be required in order to solve more difficult problems. One potential shortcoming is that the parameters used in the operators and initialization procedure were chosen arbitrarily and not optimized in any sense. Another possible place for improvement is that crossover does not attempt to match homologous genes, which probably degrades performance.

#### 3.3.2 Initialization

For the architecture portion, there are a fixed number of genes, with each gene specifying the weight for a single connection. These weights are initialized by randomly selecting for each one a real value from a uniform distribution between -2 and 2.

There are a variable number of state update genes for a genome. To initialize them, the algorithm first selects the number of such genes randomly from a range, typically between 3 and 15 inclusive. Each state update gene is then initialized as follows. The coefficient of the monomial is randomly selected as either 1 or -1. All the exponents of the monomial are set to 0 except a randomly selected one that is set to 1. (The genetic operators later introduce more complexity into the monomials. Note, however, that the operators maintain a bias towards simplicity, and even after manipulation by the operators the exponents tend to remain 0, or less likely 1 or -1.) The number of conditions is randomly selected from a geometric distribution whose decay factor is 0.3. Each condition is initialized by randomly selecting a state, plus randomly choosing real values between -1 and 1 for its upper and lower bounds.

## 4 Experiments

We now describe a sequence of experiments designed to exercise some of the capabilities of state-enhanced neural networks. Each experiment involves a different partially observable Markov decision process, or POMDP. In a POMDP, the current observations of the world do not reveal the full state of the world, i.e. there is hidden state. Since past observations and decisions provide information about the current hidden state, there needs to be some way of considering the past to assist with current decisions.

For the experiments, there are two types of POMDP problems. The first is sequence detection, which requires classification decisions based on observations of sequences of characters. Such a problem is not a typical POMDP, since the decisions do not affect the state of the world, but it still fits the definition of a POMDP because determining the current state of the sequence requires consideration of past values in the sequence.

The second, and more interesting, type of POMDP environment is a simple virtual world involving a mouse, some cheese (representing its goal state), and mazes. This environment consists of a two-dimensional rectangular grid of squares, with each square either empty, containing a wall, or containing the cheese. The mouse can occupy any of the empty squares or the goal square, and it points either up, down, left or right. The mouse senses the contents of three adjacent squares, the one straight ahead and

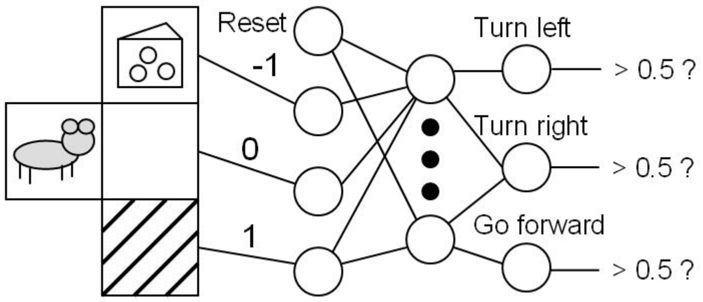


Figure 4: The control logic maps sensory inputs to control outputs.

the two diagonally ahead. To move through the maze, the mouse can turn left or right and/or advance one square ahead. When it tries to move ahead, if the square contains a wall, then the mouse remains in the current square. An external "trainer" can pick up the mouse and place it back at the beginning of either the same maze or a new maze, generally after either the mouse has found the cheese or time has expired. The mouse can sense when such a *reset* occurs. Note that a reset is different from starting a new lifetime, i.e. instantiating a new phenotype from the genome. The distinction is that after the former the mouse retains the same internal states while after the latter the states are returned to their initial values. The purpose of resets is to test memory, while new lifetimes test what happens when starting from scratch in different environments.

The mouse's "brain" contains control logic, implemented as a state-enhanced neural network, that receives inputs and produces decisions for how to move. Figure 4 shows the inputs to and outputs from the control logic. There are three sensory inputs telling the contents of the three squares it senses. The value of each 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 and 0 otherwise. Note that the mouse cannot sense to its side or behind itself (although with "memory" in the control logic, the action can depend on the contents of these squares if they were sensed previously). The three outputs of the control logic correspond to the three possible actions: turn left, turn right, and move forward. The mouse performs any of the actions for which the corresponding output is larger than 0.5, first turning and then moving. Hence, the mouse can potentially turn and move in the same step.

Note that this virtual world is similar to other two-dimensional virtual testing grounds for machine learning and intelligence. In particular, McCallum (1993) uses a nearly identical environment of a mouse in a maze searching for cheese for his work on reinforcement learning. Other similar virtual worlds include Tileworld, described by Pollack and Ringuette (1990), and Wumpus World, originally proposed by Genesereth and described by Russel and Norvig (1995).

The following is a brief summary of the problems used in the experiments:

1. Majority-Ones: The objective is to determine whether there are more ones than zeroes in a sequence to the current point. This relatively simple sequence detection problem is one for which we can analyze the internal functionality of the resulting state-enhanced neural network and how it uses internal state.
2. N-in-a-Row: The objective is to determine whether there are N (where N is 2 or 3) 1's in a row anywhere in a sequence. This is another case where we can understand and analyze the resulting network's behavior. (Such analysis is too difficult for the

other problems.)

3. $a^n b^n c^n$ Grammar: The objective is to tell the next legal terms in a sequence consisting of n a's followed by n b's and n c's. It was introduced by Schmidhuber et al. (2007) as a difficult problem to showcase Evolino, and hence is a good existing benchmark for comparison.
4. T-Maze Signal: An agent (robot or mouse) is given a signal at the start of a long corridor to determine which way to turn at the end of the corridor. Originally used for deep-memory learning by Jakobi (1998), it was later used by Gomez and Schmidhuber (2005) to demonstrate that ESP can remember the signal for an arbitrarily long time, and is hence another good benchmark for comparison.
5. T-Maze Signal Plus Counting: In this new extension of the T-maze problem, the agent does not turn immediately at the end of the corridor but rather is released into open space and must count N steps before turning. Forcing the agent to both count and recall the direction to turn increases the problem difficulty.
6. Many-Branch Maze: An agent must as efficiently as possible explore a maze with ten side branches off a main corridor to find the cheese/goal in one of them. This problem requires more complex search logic for the agent than does the T-maze.
7. T-Maze Exploration: Although this problem uses the same basic maze geometry as the T-Maze signal problem, it is different in concept. The agent must explore both possible cheese locations on its first attempt, and must "remember" the discovered location in order to navigate directly to it on subsequent attempts. This is a benchmark used by Blynel and Floreano (2003) for demonstrating evolutionary training of a CTRNN, albeit with a real robot rather than only a simulation.
8. Double-T-Maze Exploration: This is the same basic problem as the T-maze exploration problem but with more complex geometry. Blynel and Floreano (2003) presented this as a problem not fully solvable by their CTRNN approach, so we can use it to a limited extent to demonstrate the improvements provided by our approach, with the caveat that the CTRNN was applied to a real robot rather than a simulation.
9. 3-Branch Exploration: This is like the previous two exploration problems, except with three instead of two possible locations for the cheese in three different corridors. The need to remember three possible locations and act differently on each possibility increases the problem difficulty.

We now discuss the individual experiments. For each, we provide a more detailed definition of the problem, the evaluation function used by the genetic algorithm, the values of the problem-specific genetic algorithm parameters, and the results and conclusions. For the problems where there are existing results obtained by other techniques, we compare our results to the competing ones. At the end of this section describing the experiments, Table 1 summarizes the results.

## 4.1 Majority-Ones Experiment

### 4.1.1 Problem Definition

This is a sequence detection problem. The input is a sequence of an arbitrary number of zeroes and ones at the single input node of the network. At the end, the output should be one if there were more ones than zeroes in the sequence and zero if there were more zeroes than ones.

The obvious algorithmic solution to the problem is to use a counter to keep track of the difference between the number of ones and the number of zeroes observed so far. The counter is initialized to zero, and is incremented for each one observed and decremented for each zero. If the counter is positive (negative) at the end, then there were a majority of ones (zeroes). Machine learning algorithms that can naturally represent a counter (such as genetic programming with the right primitives) can find this exact solution, while other algorithms (such as ours) need to improvise something equivalent.

### 4.1.2 Evaluation Function

For the training set, we selected 500 different random sequences of binary values, 250 with a majority of ones and 250 with a majority of zeroes. For the first 400 of these sequences (200 of each type), we selected the length of each sequence to be a random value between 5 and 21 inclusive. For the next 100 sequences, we chose the length of each sequence to be a random value between 100 and 200. We included a large number of training exemplars to ensure generality. We chose a few to be moderately long sequences to ensure that the evolved solution works for longer sequences.

The test set contained 920 different sequences. The first 500 were chosen from the same distributions of sequence lengths as used for the training set. The next 400 were chosen to have random sequence lengths between 5 and 210. The final 20 had sequence lengths between 5000 and 6000. These very long sequences served to test how well the solutions generalize for test sequences much longer than the training sequences.

For a single sequence of inputs and the corresponding desired output, the error measure is defined as

$$f(e) = \left\{ \begin{array}{c} e+9 \ \text{ if } e > 0.6 \\ e+2 \ \text{ if } e > 0.2 \\ e \ \text{ otherwise} \end{array} \right\} \tag{1}$$

where $e$ is the absolute value of the difference of the desired output and the actual output. The overall score of a network is

$$\sum_{i=1}^{N_T} f(e_i) + 0.1 N_s \tag{2}$$

where $e_i$ is the error in the final output for sequence $i$, $N_T$ is the number of training exemplars (in this case 500), and $N_s$ is the number of state update genes. The term involving $N_s$ is included to encourage parsimony in the genome, i.e. to favor genomes with less state update genes.

### 4.1.3 Parameter Values

We used two different configurations corresponding to two different neural network architectures. The first was similar to the configuration used for the mouse-in-maze problems described below, employing hidden nodes and a multilayer network architecture. The second used a minimal network with a single input node directly connected to a single output node. The former is a more typical configuration and served as a good first test before proceeding to the more difficult problems. The latter yielded solutions that were easier to analyze and understand, hence providing insight into how our approach works.

For the first configuration, the population size is 2000. The network architecture is fixed with four hidden nodes, fully connected feedforward with no direct connection between the single input and single output node. So, there are eight weights and eight corresponding genes for the weights. There are four internal states including the input-output state, and therefore three free internal states. For each new input, the network is allowed two update cycles for the state changes to propagate both between nodes and within nodes, plus three extra update cycles at the end before making a final decision.

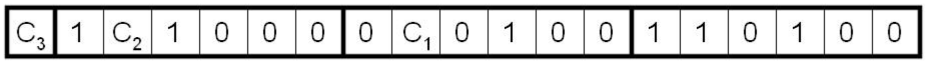


Figure 5: The genome for a sample solution to the majority-ones problem contains one weight gene and three state update genes.

For the second configuration, the population size is 5000. (Despite a simpler network architecture and solution, it is harder for the genetic algorithm to find an optimal solution, so a larger population is required than for the first configuration.) With only a single connection between the input and output nodes, only a single weight needs to be specified. There are still two update cycles per input, but only one extra cycle at the end.

Note that in this and all the other experiments there was no attempt to find the best parameters, instead using the first set of parameters that worked reasonably well. Usually this meant using the same parameters as for the last experiment and only changing them if a good solution was not found.

### 4.1.4 Results

For this experiment (and all the others), the genetic algorithm was executed ten times for each configuration. This provides a statistical sampling of performance plus evidence that any successes are repeatable.

For the first configuration, all ten times the genetic algorithm found a solution that was correct on all 500 training sequences. It required a median and a mean of approximately 6.5 pseudo-generations, or 13,000 evaluations. For seven of the runs, the best individual had three state update genes and an evaluation of 0.3. For the other three runs, the best individual had four state update genes and an evaluation of 0.4. (These were cases of the population converging to a slightly suboptimal solution and not having the diversity to escape the local minimum.) To test generalization, the optimal network from each run was executed on the test set, and nine out of the ten correctly classified all sequences in the test set. The only test case missed by the tenth was a sequence much longer than those in the training set.

For the second configuration, nine out of ten times the genetic algorithm found a solution that was correct on all training sequences. The nine successful runs required a median and a mean of approximately 40,000 and 47,000 evaluations (or equivalently, 8 and 9.4 pseudo-generations) repectively. All nine runs found solutions with three state update genes. Of the nine best-of-run networks, one correctly classified all sequences in the test set; the other eight misclassified just one test sequence, with this one sequence always being one of the very long sequences.

We draw a few general conclusions from these results. The searches were comparatively quick, which implies that this is a relatively easy problem for this approach. Using hidden nodes actually made the search faster, although as we see in the next experiment, this is not always the case. The genomes/networks found by the genetic algorithm can only be expected to classify correctly with certainty those test cases with sequence lengths within the same range as the sequence lengths for the training set. We will see what causes an occasional failure for longer sequences in the upcoming analysis of the solutions.

Examination of the optimal solutions found for the second configuration (the one with no hidden nodes) shows the same basic solution repeatedly in slightly different forms. The typical solution has a genome that looks like that shown in Figure 5, where

$C_1$, $C_2$ and $C_3$ are constants such that $C_1C_2 > 0$ and $C_3 \approx 2$. The state update genes from this genome specify state update rules given by the equations

$$(s_0)_{new} = C_1(s_1)_{old} \qquad (s_1)_{new} = (s_1)_{old} + C_2(s_0)_{old} \tag{3}$$

where $s_0$ is state 0, the input-output state, and $s_1$ is the first free internal state. Note that these state update rules use only one of the free internal states and ignore the other two (with state 1 arbitrarily chosen as the one to use).

Analysis shows that $s_1$ acts equivalently to the counter discussed in the problem description. When the input to the network is 1, the input-output state is set to $s_0 = C_3 - 1 \approx 1$; when the input is 0, $s_0 = -1$. Hence, $s_1$ is incremented by approximately $C_2$ if the input is 1 and incremented by $-C_2$ if the input is 0. Since there are two update cycles per sequence element, the increment quantities are approximately $2C_2$ and $-2C_2$ per sequence element. The final output is computed after one additional cycle during which $s_0$ is set to $C_1s_1$. Since $C_1C_2$ is positive, the final value of $s_0$ will be positive if the sequence contained a majority of ones and negative if it contained a majority of zeroes. The sigmoid transforms these values of $s_0$ to produce outputs of one and zero repectively.

This analysis of the solution provide an understanding of the occasional failure on very long sequences. If $C_3 = 2$ exactly, then $s_1$ exactly implements a counter, but with $C_3 \neq 2$, there is a slowly accumulating error proportional to $|C_3 - 2|$. In a long sequence that contains almost the same number of ones and zeroes, the accumulated error can become larger than the difference in the counts. In fact, the single solution that classified all the test cases correctly had $C_3 = 2.003$, while for all other solutions $|C_3 - 2| > 0.004$. Including very long sequences in the training set would fix this problem.

### 4.2 N-In-A-Row Experiment

#### 4.2.1 Problem Definition

Like the majority-ones problem, the N-in-a-row problem involves detection of a pattern in a temporal sequence. In this case, the pattern is a subsequence of N consecutive ones, where N is some integer. If there are N consecutive ones anywhere in the sequence, the output at the end of the sequence should be one; otherwise, the output should be zero. We perform the experiment first for the case when N=2 and then for the harder case of N=3.

As for the majority-ones problem, there exists a simple algorithmic solution using a counter. Initialize the counter to be zero. For each sequence element that is one, increment the counter by one. For each zero, if the counter is currently less than N, reset the counter to zero, and otherwise, do nothing. At the end, if the counter is at least N, then there were N consecutive ones in the sequence. We will see that our approach evolves an analogous solution using node internal state.

#### 4.2.2 Evaluation Function

For N=2, the training set contains 402 different sequences. The first 400 are randomly selected sequences whose lengths vary between 5 and 21. We ensure that half of these are positive exemplars, i.e. have two ones in a row, and half are negative exemplars. The final two exemplars are sequences of length 5000 included to ensure that the evolved network works for longer sequences. The negative exemplar consists of 5000 alternating ones and zeroes, while the positive exemplar is the same as the negative except somewhere in the middle a one is swapped with its immediately preceding zero, making two ones followed by two zeroes. The test set contains 802 exemplars. Like the training set, the first 400 exemplars have sequence length between 5 and 21; the next two exemplars are like the long sequences in the training set except with the swap in a different position; the final 400 are randomly generated sequences with lengths between 5 and 50.

For N=3, we were less thorough about training for and testing long sequences than for N=2. The training set contained 400 exemplars with sequence lengths between 5 and 21. The test set had the same distribution of exemplars.

The fitness score for a genome/individual is as given in Equation 2, and is the same as for the majority-ones problem.

#### 4.2.3 Parameter Values

The parameter values are almost identical to the ones used for the majority-ones experiment. For both N=2 and N=3, we used two different configurations, one with hidden nodes and the other without them. The one parameter that we changed was the population size, with the more challenging search problems requiring a larger population to find the optimal solution reliably. For N=2, we used a population size of 5000 for both configurations. For N=3, the population sizes were 20,000 without hidden nodes and 50,000 with hidden nodes.

#### 4.2.4 Results

There are four different sets of results.

For N=2 and no hidden nodes, all ten genetic algorithm runs produced a correct solution (i.e. a solution perfect on the training set). They required a median of about 67,000 evaluations and mean of about 75,000 evaluations to find a correct solution, and longer to compress the number of genes in the genome for a more parsimonious solution. The best solution discovered had only four genes.

For N=2 with hidden nodes, all ten runs produced a correct solution. They required a median and mean of about 60,000 and 63,000 evaluations respectively. The smallest, and therefore best, solution had four genes.

For N=3 and no hidden nodes, eight of the ten genetic algorithm runs produced a correct solution, requiring a median and mean of about 330,000 and 350,000 evaluations respectively. The best solution had four genes.

For N=3 with hidden nodes, all ten genetic algorithm runs produced a correct solution, requiring a median and mean of about 850,000 evaluations. The best solution had five genes.

All best-of-run genomes performed perfectly on their corresponding test sets.

These results indicate that the N-in-a-row problem is more difficult for our approach than the majority-ones problem. Furthermore, the use of hidden nodes only made the search more difficult for the case N=3. A possible problem is the homogeneity of the state update rules; forcing all nodes to use the same rules eliminates much of the potential power of a multinode network.

As for the majority-ones problem, the solution when there are no hidden nodes is accessible to humans and instructive to analyze. For N=2, the most compact (in terms of the number of state update genes) of the evolved solutions again uses only one of the free internal states, ignoring the other two. (Other solutions use more of the free internal states.) The equations for the state update rules are

$$(s_0)_{new} = 7.03(s_0)_{old} - 8.09(s_1)_{old} \qquad (s_1)_{new} = -0.34(s_0)_{old} + (s_1)^3_{new} \tag{4}$$

and the single weight is 2.65. Note that setting $(s_1)_{new}$ equal to $(s_1)^3_{old}$ means that $s_1$ will tend to decay to 0 when $|s_1| < 1$ but will grow rapidly when $|s_1| > 1$. So, having $|s_1|$ grow bigger than one acts like a switch after which $s_1$ will remain negative if $s_1 < -1$ and remain positive if $s_1 > 1$. When the input is 0, $s_0 = -1$, and when the input is 1, $s_0 = 1.65$. Hence, an input of 1 pushes $s_0$ quicker to -1 than an input of 0 pushes it to +1. The constants are calibrated so that it only requires two 1's to flip the switch but many more 0's. Many zeroes in a row will fool it, so again we would need to include in the training set sequences with many consecutive zeroes to ensure that the solution can handle this case.

A similar type of solution is evolved for N=3 with state update rules given by

$$(s_0)_{new} = -4.45(s_0)_{old}(s_1)_{old} - 5.87 \qquad (s_1)_{new} = 0.89(s_0)_{old}(s_1)^3_{old} - 0.47 \tag{5}$$

and a weight of 2.1.

## 4.3 $a^nb^nc^n$ Grammar Experiment

### 4.3.1 Problem Definition

The $a^nb^nc^n$ grammar problem is a difficult sequence detection problem introduced by Schmidhuber et al. (2007). It provided a way to demonstrate the power of the Evolino technique, since the problem was beyond the capabilities of neural-network techniques other than Evolino and its predecessor, LSTM.

The problem is based on a simple grammar involving five letters: $a$, $b$, $c$, $s$ and $t$. A legal sequence in this grammar starts with $s$, then has $n$ $a$'s, $n$ $b$'s, $n$ $c$'s, and concludes with a $t$, where $n$ is any non-negative integer. Examples of legal sequences are $st$, $sabct$, and $saaabbbccct$. The objective for the neural network is, at any point in the sequence, to tell which letters are legal to occur next. For instance, if the sequence so far is $saa$, then the possible next letters are $a$ and $b$.

This problem is difficult for a neural network because it requires keeping count of the number of $a$'s as they are added to the sequence and using this count to determine when $b$'s should transition to $c$'s and then when $c$'s should transition to the final $t$.

### 4.3.2 Evaluation Function

We define there to be four inputs and four outputs. There is an input corresponding to each of the letters $s$, $a$, $b$ and $c$, with the input high if that letter is the current term in the sequence and low otherwise. (Since $t$ is always the last term, it is never an input to predict the next term.) There is an output corresponding to each of the letters $t$, $a$, $b$ and $c$, with the correct output high if that letter is legal as the next term in the sequence and low otherwise. (Since $s$ is always the first term, it is never a legal next term.)

We use eleven training cases, one for each $n$ between 0 and 10 inclusive. For each training case, there are $3n + 2$ terms in the sequence and hence $3n + 1$ different steps after which to determine the possible next terms. The evaluation function is the sum of two separate subscores, one for all the desired outputs equal to 0 and the other for all the desired outputs equal to 1. We separate these subscores because there are many more 0 outputs than 1 outputs, and we do not want to reward always selecting a 0. Each subscore is computed according to Equation 2.

### 4.3.3 Parameter Values

We used a network architecture with four hidden nodes. The nodes had two internal states. The population size was 20000.

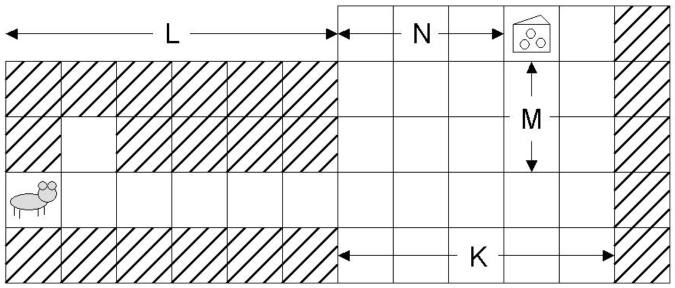


Figure 6: This is the geometry of the maze for the problems in Sections 4.4 and 4.5. For the former, N=0, K=1 and M=5; for the latter, N=3, K=7 and M=9.

### 4.3.4 Results

Of all the problems investigated, this was the most difficult one for the search algorithm to find a solution. Of the ten genetic algorithm runs, only one produced a solution that solved all ten test cases correctly. This run required 115 pseudo-generations, which is roughly 2.3 million evaluations. The solution it found had five state update genes.

There was no generalization to exemplars that were not part of the training set, i.e. the evolved solution was not correct for $n > 10$. Evolino had far superior generalization, as it was able to find a solution that generalized to $n = 53$. This presents a clear future challenge to improve the generalization capability of our approach on sequence detection problems. However, the problems of greater interest to us are the mouse-in-maze problems, since they are more along the path towards our ultimate goal of adaptive physical agents. We now examine a set of experiments involving this type of problem.

## 4.4 T-Maze Signal Experiment

### 4.4.1 Problem Definition

The T-maze signal problem has been previously used by multiple researchers, including Jakobi (1998), Bakker et al. (2003) and Gomez and Schmidhuber (2005), to investigate reinforcement learning with memory. Hence, it provides a comparison with previous approaches.

The maze consists of a long corridor with the agent (for us, a mouse) starting at one end and a T-junction at the other end. When the mouse reaches the T-junction, it must turn either right or left to reach the goal/cheese. The information about which direction to turn is provided at the start of the corridor by some type of signal, which we have implemented as a gap in the wall on the side to which the mouse should turn. The control algorithm needs to learn to recognize the significance of the signal and to remember it for the entire length of the corridor.

Figure 6 illustrates a generalized version of this maze geometry that covers the T-maze-with-counting problem as well as this one. For this problem, N=0, K=1 and M=5, which means that there is no choice besides turning left or right at the end of the corridor. The length of the corridor, L, can vary, and ideally the control algorithm should work for any length L. Note that in Figure 6, the gap and the cheese are both to the mouse's left, but the control law must also handle the case when both are on the right.

#### 4.4.2 Evaluation Function

The training set consists of six different mazes, three with the goal on the left and three symmetric cases with the goal on the right. The three different values for the corridor length, L, are 5, 15, and 30.

For each maze in the training set, the mouse starts at the beginning and goes until it either reaches the cheese or takes the maximum number of steps, which we set to 40. The penalty incurred by the mouse navigating a maze is the sum of the following four terms

- the total number of steps the mouse takes
- the number of times it attempts to walk into a wall
- a fixed value, in this case 30, if it does not find the cheese
- the number of steps it is from the cheese at the end

The overall score is the sum of the penalties for each of the six test cases.

#### 4.4.3 Parameter Values

The configuration is similar to those used with hidden nodes for the majority-ones and N-in-a-row problems with the following differences. The population size is 20,000, there are five hidden nodes (rather than four) in the fixed network architecture, and there are two available free internal states (rather than three).

#### 4.4.4 Results

There are two definitions of success. One is that the mouse reaches its goal in all test cases before time expires. The second, more stringent, definition is that the solution is optimal, i.e. it takes the minimum number of steps to reach its goal in all test cases. An optimal solution cannot waste a single move; for example, it must be able to turn and move in the same time step at the end of the corridor (as opposed to using one step to turn and the next to move).

Nine out of ten genetic algorithm runs found a solution that reached all the goals, and seven out of ten found solutions that were optimal on the training set. The seven runs required a median of 240,000 evaluations and mean of 320,000 to reach optimal solutions. The most compact solution had three state update genes.

We tested the seven optimal solutions for a corridor of length 10,000 (i.e. L=10000), which is much longer than any of the training cases. All seven solutions performed optimally in this case also, hence demonstrating generalization to corridors of far greater length and the ability to remember the initial signal for very long times. Gomez and Schmidhuber (2005) showed that ESP, like our approach but unlike other recurrent neural networks, could remember the signal for very long times. Therefore, our approach provides performance equivalent to the best of its competitors.

### 4.5 T-Maze Signal with Counting Experiment

#### 4.5.1 Problem Definition

This problem is an extension of the T-maze signal problem that is more difficult. It requires the agent/mouse not only to remember to which side the signal indicated to turn but also to count the number of steps after the end of the corridor before turning. We have invented this problem to highlight the enhanced capabilities of our approach.

As shown in Figure 6, when $K >> 1$, the mouse does not run into a wall at the end of the corridor but is instead released into open space. The goal state can be reach by

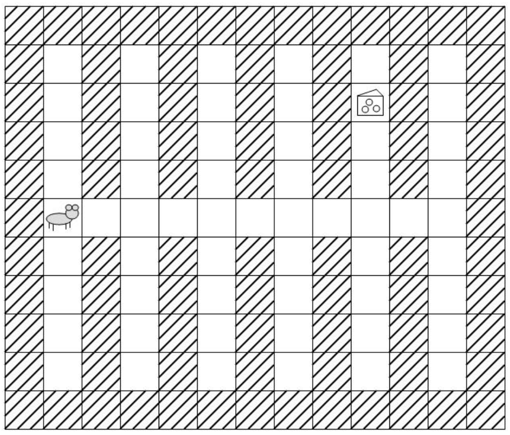


Figure 7: In the many-branch problem, the goal is in one of the ten side corridors not adjacent to the starting position. It is always three steps from the main corridor.

proceeding N steps beyond the end of the corridor and then turning to the side of the signal. When N>1, this requires counting the number of steps beyond the end before turning. For any given instance of the problem, N is fixed, as is M and K. For this experiment, we used N=3, K=7 and M=9. An optimal solution to the problem requires the mouse to remember both the side of the signal at the beginning of the corridor and the number of steps taken beyond the end of the corridor.

### 4.5.2 Evaluation Function

The training instances are the equivalent six mazes as for the T-maze signal problem described in Section 4.4. The penalty function is also the same. The maximum number of steps is raised to 50.

### 4.5.3 Parameter Values

The parameters are the same as for the T-maze signal problem except that the population size is now 50,000.

### 4.5.4 Results

Ten out of ten runs found a solution that reached all the goals, and eight out of ten found an optimal solution. For a corridor of length 10,000, seven of the eight performed optimally in this case also, hence displaying good generalization. The eight runs required a median and mean of 1,250,000 evaluations to reach solutions that were optimal. The most compact solution had three state update genes.

## 4.6 Many-Branch Experiment

### 4.6.1 Problem Definition

As shown in Figure 7, the maze consists of a main corridor with ten different passageways branching off of it. The cheese can be in any of the ten side passages. Each side passage is six steps deep (unlike in the figure that shows them only four steps deep), with the cheese always exactly three steps in. Since the mouse can determine whether the cheese is in a passageway after two steps, an efficient search should not continue to the end of each side passage but rather turn around after two steps if the cheese is not there. Note that this problem requires a simple maze search strategy combined with

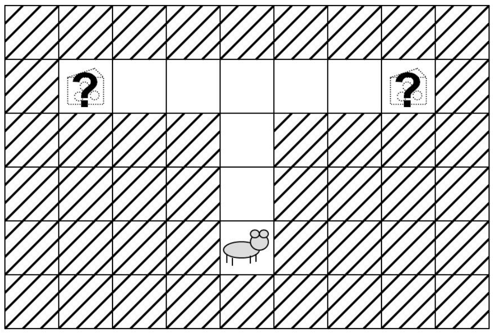


Figure 8: In the T-maze exploration problem, the objective is to discover the cheese location on the first attempt and to remember and utilize this information on subsequent attempts.

the ability to count steps and act accordingly. Hence, it provides a test of the ability to evolve moderately complex behaviors.

#### 4.6.2 Evaluation Function

There are ten test cases, each with the cheese in a different side passage. The score is computed the same as for the T-maze signal problems, except the maximum number of steps is 110. To reach the goal in each of the test cases within 110 steps requires a search strategy that only enters partway into each side passage, thus providing strong incentive to find such a strategy.

#### 4.6.3 Parameter Values

The setup is the same as the T-maze signal problems, with a population size of 50,000.

#### 4.6.4 Results

In nine of the ten runs, the discovered solution could find the cheese in any of the ten possible locations, checking each passageway and turning around if it did not detect the cheese after two steps. None of the solutions were fully optimal, as all explored either clockwise or counter-clockwise rather than checking the passageways on both the left and right as proceeding down the main corridor. However, all nine of the solutions were optimal in the sense of wasting no steps in the process of turning and moving. This is actually tricky; e.g., to turn around, the mouse's first step must be just a left or right turn (a move forward draws a penalty for hitting wall) and the next step a combination of a turn and a move. The nine successful runs required a median of 700,000 evaluations and mean of 800,000 evaluations to find their solutions.

### 4.7 T-Maze Exploration Experiment

#### 4.7.1 Problem Definition

The T-maze exploration problem was utilized by Blynel and Floreano (2003) to validate their use of an evolutionary algorithm for training CTRNNs. It tests the ability of the approach to find a control algorithm that adapts its behavior based on knowledge gained about the maze. Blynel and Floreano (2003) used a real robot in their work, while our work uses a simulation, so the results are not fully comparable. (The noise of real sensors and actuators in general makes the control of a real robot more difficult than the control of a simulated one.)

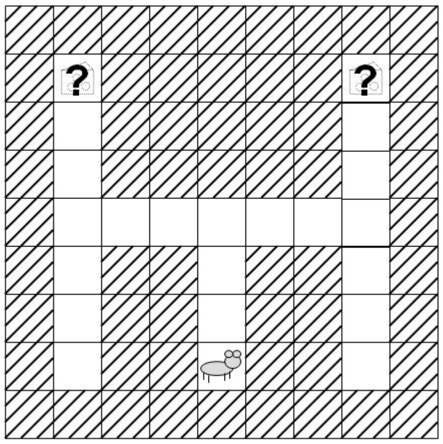


Figure 9: While the double T-maze presents a more complex geometry than the simple T-maze, the exploration problem is the same.

The maze geometry differs from that of the T-maze signal problem in that there is no signal at the beginning of the corridor and the corridor is shorter. The agent/mouse performs three attempts at navigating the maze, and the cheese is placed in the same location for each attempt. This location can be in either of two possible spots, one in each branch. On the first attempt, the mouse should find the cheese whether it is in the left or right branch. On subsequent attempts, the mouse should go directly to the cheese based on its "memory" of what occurred in the first attempt. This is illustrated in Figure 8.

### 4.7.2 Evaluation Function

There are two test cases, one for each possible location of cheese. For each test case, there are three successive attempts to find the cheese. The first two attempts are allowed a maximum of 20 steps The third attempt is allowed a maximum of 10 steps, which prevents the mouse from doing any exploration on this attempt. The score is computed the same as for the other maze problems.

### 4.7.3 Parameter Values

The parameter values are the same as for the other maze problems, with a population size of 50,000.

### 4.7.4 Results

In all ten genetic algorithm runs, the discovered solution was essentially correct, i.e. it found the cheese on all three attempts for both test cases. In six of the ten runs, the solution was optimal in that there were no wasted movements. In the other four cases, on the first attempt if the cheese was not in the first branch explored, the mouse would go to the end of the branch before turning around rather than turning around as soon as it did not detect the cheese. The optimal runs required a median of 1,500,000 evaluations and a mean of 1,900,000 evaluations.

### 4.8 Double-T-Maze Exploration Experiment

#### 4.8.1 Problem Definition

The double-T-maze exploration problem was proposed by Blynel and Floreano (2003) as a more difficult version of the T-maze exploration problem to provide a greater challenge to their CTRNN technique and others. Their CTRNN approach could not fully solve the problem, so it is a good one for demonstrating the benefits of state-enhanced neural networks. As for the T-maze exploration problem, Blynel and Floreano (2003) used a real robot rather than a simulation, which usually makes the problem more difficult; therefore, this should not be considered a head-to-head comparison between CTRNN and our approach.

As in the simple T-maze exploration problem, there are two possible locations for the goal/cheese. The agent/mouse must check both these locations on the first attempt and then use this information to go directly to cheese on subsequent attempts. The difference is that the geometry of the double T-maze, which is pictured in Figure 9, is more difficult to navigate. It requires multiple turns to travel between the starting location and either potential cheese location, as well as between the two possible cheese locations.

When seeded with a solution to the simple T-maze, the evolutionary seach algorithm of Blynel and Floreano (2003) could find a CTRNN that could find the goal location in all but one case. If the goal location was not the possible location explored first during its first attempt, it could not navigate to the second location.

#### 4.8.2 Evaluation Function

The evaluation function is the same as for the T-maze exploration problem except that the maximum steps allowed are 35 for the first attempt and 15 for subsequent attempts.

#### 4.8.3 Parameter Values

The parameter values are the same as for the simple T-maze exploration problem, including the population size of 50,000.

#### 4.8.4 Results

In eight of the ten genetic algorithm runs, the discovered solution was essentially correct. None of the solutions was optimal in that there was always some small wasted movements, usually on the first attempt proceeding to the end of the first corridor rather than immediately turning around at the the point when the cheese is clearly not present. The runs yielding correct solutions required a median of 2,750,000 evaluations and a mean of 3,000,000 evaluations.

Unlike CTRNN, our state-enhanced neural network approach consistently discovers a solution that can find the cheese on all three attempts whether the cheese is in the left or the right branch. (We reiterate the caveat that CTRNN was used with a real robot and our approach with a simulated robot.) In particular, on the first attempt, if the cheese is not in the first location explored, the control algorithm can navigate efficiently from the first possible location to the second.

### 4.9 3-Branch Exploration Experiment

#### 4.9.1 Problem Definition

This is a problem that we have invented that is similar to the T-maze exploration problem but adds an extra level of difficulty. Instead of two possible locations for the goal/cheese, there are three possible locations in three different branches, as shown

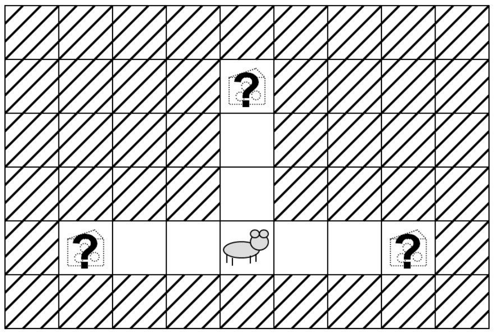


Figure 10: The 3-branch exploration problem differs from the other exploration problems in that there are three possible locations for the cheese instead of just two.

in Figure 10. The agent/mouse must explore the three possible locations for the cheese on its first attempt and proceed directly to the right location on subsequent attempts.

#### 4.9.2 Evaluation Function

There are three test cases, each with the cheese in a different location. The score is computed the same as for the previous exploration problems, with three attempts evaluated for each test case. The maximum number of steps is 30 on the first attempt and 7 on the other two attempts.

#### 4.9.3 Parameter Values

The parameter values are the same as for the T-maze exploration problems, including the population size of 50,000.

#### 4.9.4 Results

In seven of the ten genetic algorithm runs, the discovered solution was essentially correct, finding the cheese in any of the three locations on the first attempt and proceeding to the cheese without testing the other two possible locations in the next two attempts. None of the solutions was optimal in that there were no wasted movements. The runs yielding correct solutions required a median of 2,500,000 evaluations and a mean of 2,800,000 evaluations.

## 5 Conclusion and Future Work

In this paper, we examined a new type of neural network, a state-enhanced neural network, whose nodes possess multidimensional internal state with selectable and potentially complex internal dynamics. Providing state for the nodes' computation process allows the nodes to use memory of the past in their current computations. We demonstrated that an evolutionary algorithm can find a genome specifying internal dynamics as a set of state update rules that are suited for a particular task and/or environment.

Furthermore, we showed how this approach achieves performance on a suite of deep-memory POMDPs comparable to that of the best current neural network solutions. Our technique solved three benchmarks designed to showcase three of the best neural network approaches, as well as a fourth, previously unsolved benchmark. We also introduced three new mouse-in-maze POMDP problems designed to further challenge this class of techniques and showed that our approach could solve these also.

| Problem Name | Population | Evaluations | Correct | Optimal |
|---|---|---|---|---|
| Majority-Ones | 2,000 | 13,000 | 10 | 9 |
| 2-in-a-Row | 5,000 | 63,000 | 10 | 10 |
| 3-in-a-Row | 50,000 | 850,000 | 10 | 10 |
| $a^n b^n c^n$ Grammar | 20,000 | 2,300,000 | 10 | 0 |
| T-Maze Signal | 20,000 | 320,000 | 9 | 7 |
| T-Maze Signal Plus Counting | 50,000 | 1,250,000 | 10 | 7 |
| Many-Branch Maze | 50,000 | 800,000 | 9 | 0 |
| T-Maze Exploration | 50,000 | 1,900,000 | 10 | 6 |
| Double-T-Maze Exploration | 50,000 | 3,000,000 | 8 | 0 |
| 3-Branch Exploration | 50,000 | 2,800,000 | 7 | 0 |

Table 1: This is a simple summary of the parameters and results of the experiments. There were ten genetic algorithm runs performed for each problem. The evaluations column contains the mean number of individuals evaluated before convergence to a correct solution. The correct column contains (a) for sequence detection problems, the number of runs yielding solutions correct on the entire training set and (b) for mouse-in-maze problems, the number that correctly solved the maze within the time limits. The optimal column contains (a) for sequence detection problems, the number or runs producing a solution correct on the entire test set and (b) for mouse-in-maze problems, the number with no wasted movements and which, where applicable, generalized to new mazes correctly.

Indeed, the aggregation of a broad set of problems that could serve as a standard to evaluate deep-memory learning techniques is a secondary achievement of our work, in addition to the primary achievement of developing a new technique and demonstrating some of its capabilities.

While this work represents progress, it is still just preliminary, with a variety of ways to potentially expand its power. One possible enhancement is allowing heterogeneity among the nodes with respect to internal dynamics, and hence node functionality. In biological organisms, all cells have the same genome; heterogeneity results from different cells expressing different genes. The same could be the case in state-enhanced neural networks if the position/role of a node were represented as an internal state (or set of states) and this state could be used in the regulator portion of genes.

A more distant goal is to include in the genome genes controlling the network architecture and learning algorithm, such as those from the GRN work described in Section 2.2. Such an enhanced genome would allow the simultaneous evolution of network structure, node functionality, and weight adjustment technique, as discussed by Montana et al. (2006). Furthermore, there could be coupling between the states, so architectural state could drive computation and learning and vice versa. In practice, the difficulty is that an enhanced genome would result in a very large search space. We would need to find better approaches to searching this space, with one possible improvement being to build on already known solutions rather than starting from scratch each time (as advocated by D'Silva et al. (2005) in a different context).

## References

Aberdeen, D. (2003). A (revised) survey of approximate methods for solving partially observable markov decision processes. Technical report, National ICT Australia, Canberra, Austalia.

Abraham, A. (2004). Meta learning evolutionary artificial neural networks. *Neurocomputing*, 56:1–38.

Aihara, K., Takabe, T., and Toyoda, M. (1990). Chaotic neural networks. *Physics Letters A*, 144(6):333–340.

Astor, J. and Adami, C. (2000). A developmental model for the evolution of artificial neural networks. *Artificial Life*, 6(3):189–218.

Asuncion, A. and Newman, D. (2007). UCI machine learning repository. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

Bakker, B., Zhumatiy, V., Gruener, G., and Schmidhuber, J. (2003). A robot that reinforcement-learns to identify and memorize important previous observations. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2003)*, pages 430–435.

Baxter, J. (1992). The evolution of learning algorithms for artificial neural networks. In Green, D. and Bossomaier, T., editors, *Complex Systems*. IOS Press, Amsterdam.

Blynel, J. and Floreano, D. (2003). Exploring the t-maze: Evolving learning-like robot behaviors using ctrnns. In *Proceedings of the 2nd European Workshop on Evolutionary Robotics*, pages 593–604.

Bongard, J. (2002). Evolving modular genetic regulatory networks. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 1872–1877.

Cangelosi, A., Parisi, D., and Nolfi, S. (1994). Cell division and migration in a 'genotype' for neural networks. *Network*, 5:497–515.

Chalmers, D. (1990). The evolution of learning: An experiment in genetic connectionism. In *Proceedings of the 1990 Connectionist Models Summer School*, pages 81–90.

Dellaert, F. and Beer, R. (1996). A developmental model for the evolution of complete autonomous agents. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 393–401.

D'Silva, T., R, J., Chrien, M., Stanley, K., and Miikkulainen, R. (2005). Retaining learned behavior during real-time neuroevolution. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.

Eggenberger-Hotz, P., Gomez, G., and Pfeifer, R. (2002). Evolving the morphology of a neural network for controlling a foveating retina and its test on a real robot. In *Proceedings of the Eighth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life VIII)*, pages 243–251.

Floreano, D., Durr, P., and Mattiussi, C. (2008). Neuroevolution: From architectures to learning. *Evolutionary Intelligence*, 1(1):47–62.

Floreano, D. and Mattiussi, C. (2001). Evolution of spiking neural controllers for autonomous vision-based robots. In Gomi, T., editor, *Evolutionary Robotics: From Intelligent Robotics to Artificial Life*. Springer, Tokyo.

Funahashi, K. and Nakamura, Y. (1993). Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801–806.

Gers, F. and Schmidhuber, J. (2001). Lstm recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340.

Gerstner, W. and Kistler, W. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gomez, F. and Miikkulainen, R. (1999). Solving non-markovian control tasks with neuroevolution. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1356–1361.

Gomez, F. and Schmidhuber, J. (2005). Co-evolving recurrent neurons learn deep memory POMDPs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.

Gruau, F. (1995). Automatic definition of modular neural networks. *Adaptive Behavior*, 3(2):151–183.

Harp, S., Samad, T., and Guha, A. (1989). Towards the genetic synthesis of neural networks. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 360–369.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hussain, T. (2004). Generic neural markup language: Facilitating the design of theoretical neural network models. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 235–242.

Ichinose, N., Aihara, K., and Kotani, M. (1993). An analysis on dynamics of pulse propagation networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 2315–2318.

Izhikevich, E. (2004). Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070.

Jakobi, N. (1995). Harnessing morphogenesis. In *International Conference on Information Processing in Cells and Tissues*, pages 29–41.

Jakobi, N. (1998). Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6(2):325–368.

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

Kumar, S. and Bentley, P. (2003). *Computational Embryology: Past, Present and Future*, pages 461–477. Springer-Verlag, New York.

Luke, S. (2002). An evolutionary computation and genetic programming system. http://cs.gmu.edu/ eclab/projects/ecj/docs/.

McCallum, A. (1993). Overcoming incomplete perception with utile distinction memory. In *International Conference on Machine Learning*, pages 190–196.

Montana, D. and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 762–767.

Montana, D., VanWyk, E., Brinn, M., Montana, J., and Milligan, S. (2006). Genomic computing networks learn complex POMDPs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 233–234.

Nolfi, S., Miglino, O., and Parisi, D. (1994). Phenotypic plasticity in evolving neural networks. In *Proceedings of the Internation Conference from Perception to Action*, pages 146–157.

Osana, Y., Hattori, M., and Hagiwara, M. (1996). Chaotic bidirectional associative memory. In *IEEE International Conference on Neural Networks*, pages 816–821.

Pearlmutter, B. (1995). Gradient calculation for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228.

Pollack, M. and Ringuette, M. (1990). Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189.

Russel, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Schmidhuber, J., Wierstra, D., Galiolo, M., and Gomez, F. (2007). Training recurrent networks by evolino. *Neural Computation*, 19(3):757–779.

Schmidhuber, J., Wierstra, D., and Gomez, F. (2005). Evolino: Hybrid neuroevolution / optimal linear search for sequence learning. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 853–858.

Stanley, K. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Stanley, K. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130.

Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Whitley, D. (1989). Applying genetic algorithms to neural network learning. In *Proceedings of the Seventh Conference of the Society of Artificial Intelligence and Simulation of Behavior*, pages 137–144.

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

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.