ANOTHER ONE BITES THE DOT: TEACHING A NEURAL NETWORK TO PLAY PAC-MAN USING BIOLOGICALLY MOTIVATED LEARNING TECHNIQUES

BY
DANIEL MATTHEW GOLDSMITH

A Thesis

Submitted to the Division of Natural Sciences
New College of Florida
in partial fulfillment of the requirements for the degree of
Bachelor of Arts in Computer Science
Under the Sponsorship of Karsten Henckell

Sarasota, Florida
May, 2010

Table of Contents

Table of Figures

ANOTHER ONE BITES THE DOT: TEACHING A NEURAL NETWORK TO PLAY PAC-MAN USING BIOLOGICALLY MOTIVATED LEARNING TECHNIQUES

Daniel Goldsmith

New College of Florida, 2010

ABSTRACT

A branch of Artificial Intelligence known as natural computing, which proposes learning algorithms based on naturally occurring processes such as natural selection and biological processes such as neural networks, has been shown to be capable of learning and generalization of complex problems. This thesis evaluates two such biologically motivated techniques – temporal difference learning, which learns by reinforcement, and a genetic algorithm, which learns by natural selection – by teaching a neural network to play the classic arcade game *Pac-Man*. A replica of *Pac-Man* was created using the Python programming language, and networks trained with temporal difference learning, a genetic algorithm, and a combination of the two, each for 32 hours, were shown to exhibit concrete strategies and a significant improvement compared to a random solution.

Karsten Henckell

Division of Natural Sciences

1. Introduction

When early computer researchers began to study what would become known as Artificial Intelligence, their goal was to build a "strong AI" – essentially, a self-aware, intelligent computer.[1] Rather than turning to the only known successful implementation of intelligence – the human brain – and attempting to understand how it produces intelligent behavior from simple internal processes, early AI researchers attempted to create intelligence by discovering and simulating every known "feature" of intelligence.[2] One such example is understanding natural language, a problem that, more than fifty years later, researchers still struggle with. In more recent years, however, artificial intelligence researchers have returned to the roots of intelligence by pursuing methods that are inspired by biological processes, a field known as "Natural Computing."[3]

In nature, life evolved from simple unicellular organisms to today's diverse populations of intelligent, self-aware animals. This process of evolution, driven by genetics and the process of natural selection, provides a method for species to constantly improve over many lifetimes. While some behaviors are evolved, for most species most actual behavior is learned throughout the lifetime. The animal brain, made up of anywhere from hundreds to millions of neurons, is capable of complex decision-making and learning. Learning *capability* comes from diversity and selection over thousands of lifetimes in evolution, but the learning itself takes place during an individual's lifetime in the brain.

Some methods that attempt to mimic known biological processes (such as the brain, evolution, or even particular behaviors such as insect swarms and bird flocking) have been successful at learning to do tasks that are difficult to do with traditional artificial intelligence techniques, such as pattern recognition, especially with noisy data. What's more, many of these systems can learn to do

[1] (Jones 2008) 4
[2] Ibid 5
[3] Ibid 10

one task and automatically adapt themselves to similar tasks effectively with little or no human intervention. While traditional artificial intelligence systems, such as rule-based and logic-based systems, neural networks with supervised learning, etc., rely heavily on human input to create the rules and provide correct output, some newer systems are capable of learning without significant human input (for example, with feedback only from sensors connected to the system). Such systems rely on the theory of reinforcement learning. Reinforcement learning trains the system to take actions which give it the highest reward in the long run. Since the reward is simply the feedback from the environment (in the game it could be a score, or for a robot it could be its success at a particular task like walking), no human intervention is required during training other than the experimental setup.

In video games, the traditional approach is essentially the standard. Non-player characters in video games approximate intelligent action by using simple rule-based systems and small finite state machines.[4] While this works for many games, the simulation of intelligence is fairly "thin" – players can often exploit minor loopholes in the artificial intelligence scheme to gain a major advantage. Even worse, the human can use the exact same tactic every time, but the artificial intelligence often has no way of adapting to the new behavior.

It is this last capability – the ability to learn from and adapt to an opponent's choices – that makes humans superior to computers when it comes to virtual environments that share many similarities with realistic simulations. In this thesis, I will examine a popular but relatively simple video game, Pac-Man, which provides both a reasonably complicated and dynamic environment as well as several direct ways of analyzing the overall capability of the agent[5]. To create an agent capable of learning to play this game, I will examine and combine several biologically motivated methods in artificial intelligence. The first of these is the artificial neural network, a simple model based on the

[4] In the past, when CPU cycles were scarce and graphics rendering took the vast majority of the computer's time, these simple solutions were a necessity. As the number of CPU cycles and cores increases, however, it becomes possible to offload AI processing into a different core, which allows more complicated processes – such as neural networks – to run in real-time with the game itself.

[5] The game will be analyzed in more detail in the next chapter

structure of the human brain. I will be using a version of the well-known multi-layer Perceptron, modified to allow for time-based feedback connections. To train the network, and allow it to learn during play, I will use reinforcement learning techniques. Finally, to optimize the network structure, I will use an evolutionary algorithm, a model of the biological process of evolution. Previous researchers have taught neural networks using either a genetic algorithm or reinforcement learning and shown positive results in both cases. Many of those solutions are demonstrated, however, on simple games (for example, board games like chess), on purpose-built environments that make large sacrifices in realism in order to obtain simplicity, or on simplified versions of more complicated games (for example, some studies learn to play a highly simplified version of Pac-Man). In this study, I use a "clone" of Pac-Man – a piece of software I wrote to closely simulate the original arcade version – as the learning environment for a program that learns using all three techniques: artificial neural networks, reinforcement learning, and a genetic algorithm. By using an environment designed for human players, we hope to show that artificial intelligence techniques that are closely based on relevant biological systems are capable of building an effective learning agent for playing complicated video games in a limited amount of time.

2. Background

2.1 Artificial Neural Networks

2.1.1 The Natural Brain

At the lowest level, the human brain is characterized by billions of interconnected neurons. Each individual neuron has a single projection called an axon, which may extend over a meter in length, and thousands of dendrites, which branch and narrow as they extend, generally only a few micrometers, from the body of the neuron. Each of a neuron's dendrites connects to another neuron's axon, forming a synapse, and a single axon may host thousands of connections, implying many trillions of total connections throughout the human brain. When activated, either by impulses received from other neurons or from external sources (for example, sensory organs), the neuron will send an impulse across the axon.[6] Many dendrites may connect to an axon, allowing many neurons to receive the impulse from one neuron. Synapses are either excitatory or inhibitory – they either increase or decrease the voltage received by the receiving neuron. The strength of the synapse determines how strong the excitatory or inhibitory effect is. This strength can change often, with a change lasting anywhere from a few seconds to a lifetime. These longer-term changes, caused by repeated activations, form the basis of synaptic plasticity, which is believed to be the basis of learning and memory.[7]

At a higher level, the seat of human intelligence is the neocortex, which is organized into six layers. While it was initially believed that signals only propagated away from sensory input, neuroscientists have since found that feedback connections play an important role in neurobiology.[8]

[6] Depending on the type of neuron, the impulse may be electrical, in the case of fast-propagating signals, or chemical, in the case of more complicated messages
[7] (de Castro 2006) 127
[8] (Gilbert and Sigman 2007)

While most scientists do not agree on the exact function or purpose of these connections, it has been suggested that such connections allow neurons in higher layers to excite earlier neurons in anticipation of an expected input. This allows the brain to compare expectation with reality – when expectation differs (or is nonexistent), as is often the case, synapses adapt to improve accuracy on future predictions, and learning occurs.[9] In fact, Jeff Hawkins, a computer scientist who actively works in the field of artificial intelligence, has argued that this predictive function underlies all aspects of human intelligence and is critical to building artificial intelligence systems that hope to match or exceed human intelligence. What's more, this predictive function is inherently based on patterns that occur in time; these predictions are based on patterns of actions and sensations that occur at discrete times. Artificial intelligence mechanisms that capture the spirit of these temporal predictions, he predicts, will bring us closer to realizing a truly intelligent machine.

2.1.2 The Artificial Neural Network

An Artificial Neural Network, taking inspiration from the natural brain, is made up of a number of nodes, also called artificial neurons, which are connected amongst one another. These artificial neurons are often arranged into layers. The first layer usually acts as input, while the last layer generally represents the output of the network. Each artificial neuron is a two-part machine. First, it sums all inputs from its dendrites. These inputs will either be from outside the network or will be from other neurons that are connected to this neuron. Second, it passes this sum into an activation function. Early versions of the artificial neuron used a "step" function, or threshold, to determine activation – once a neuron received enough activation from other neurons, it would fire. Modern multi-layer networks learn based on a rule that requires a differentiable function. To retain a step-like function (the type of activation used by biological synapses), but satisfy differentiability, modern neurons use sigmoid-like functions (such as the hyperbolic tangent or the logistic function). The artificial neuron sends this output across all of the dendrites connected to it. Each dendrite has a

[9] (Hawkins 2004) 113

weight – similar to the synaptic strength – which is modified during training to allow the network of neurons to learn. When a signal passes into a dendrite, the connection passes out the signal multiplied by the weight.

The most common type of artificial neural network is the Multi-Layer Perceptron (MLP). The original Perceptron, created in 1957 by Frank Rosenblatt at Cornell University, served as a simple linear classifier (that is, it was capable of separating input into one of two categories). The Perceptron consists of two layers of neurons, a layer of inputs fully connected to a layer of outputs. Each connection between neurons contains a weight; by changing these weights according to a training algorithm, the Perceptron would learn to classify (simulating synaptic plasticity). To pass a value from one layer to the next, each neuron sums its inputs, passes them through a linear activation function, and sends that value along its axons. When a value passes through an axon, it is multiplied by the axon's weight. Each neuron in the following layer will then receive a different value from the initial neuron depending on the weights of its dendrites.

To learn, the Perceptron requires a "supervisor" who knows the correct output for a portion of the possible inputs. Given a set of inputs, as well as the corresponding correct outputs, the Perceptron could learn to correctly classify the inputs by modifying each weight (**w**$_i$) according to the Delta Rule:

$$\boldsymbol{w_i} = \boldsymbol{w_i} + \boldsymbol{\alpha} \times \boldsymbol{g_i}$$

The learning rate, **α**, is a small number that slows training so that the Perceptron is able to generalize to inputs not in the training set.[10] The gradient (**g**) for the delta rule is calculated as follows:

$$g_i = (expected_i - actual_i) \times \ input_i$$

Essentially, the error (expected output minus actual output) is multiplied by the input. The gradient gives the magnitude and sign of the weight change needed to reduce the error the most.

[10] (Jones 2008) 260

This type of error reduction is known as "gradient descent." If the n weights of the Perceptron are considered as a point in n-dimensional space, we can add one more dimension which represents the error for those weights. This n+1-dimensional function is called the error gradient. Whenever it reduces the error by changing the weights, the delta rule and other gradient decent methods move down this gradient by a small amount in the direction where the slope is the most negative – that is, where the immediate error is reduced the most. The Perceptron was very limited, however: it was only capable of solving linearly separable problems.[11] Even a very simple problem, such as the logical exclusive or ("XOR"), which takes just two binary inputs and has one binary output, cannot be learned by a Perceptron.

In response to this problem, the multi-layer Perceptron was created. At first, even a Perceptron with multiple layers could not solve the problem because there was no effective way to train it. The issue at hand involves discovering how to assign error to internal neurons (error at the output layer, of course, is simply the difference between the desired and the actual output). The backpropagation algorithm showed that it was possible to train a multi-layer Perceptron by propagating the error at each output over its dendrites towards the neurons in the preceding layers. Rather than using a linear activation function (as the Perceptron allows), backpropagation requires that the neurons have a differentiable, nonlinear activation function. The backpropagated error is then used in the generalized delta rule (often just called "the delta rule" since standard Perceptrons are no longer in wide use), which uses a modified definition of the error gradient **g**:

$$\boldsymbol{g} = f'(neuronActivation) \times errorSignal$$

The slope of the gradient is the derivative of the activation function, *f'*, applied to the activation level of the neuron. The errorSignal is the sum of errors backpropagated to the neuron.

[11] (Jones 2008) 257

Mathematically, the multi-layer Perceptron with *n* input neurons and *m* output neurons represents a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$. Given a multi-layer Perceptron with at least two hidden layers, any function which maps n inputs to m outputs can be represented to any degree of accuracy so long as there are enough neurons (one hidden layer is only suitable to distinguish any continuous function).[12] This is obviously a powerful model; however, it suffers from several drawbacks. First and foremost, it can be trained only with an expert supervisor who can provide correct answers for a representative sample of the possible inputs. In a video game, it is almost never known what the correct action is. For a board game like backgammon or chess, a database of master games could be used. Unfortunately, while master games undoubtedly represent good play, mistakes could lead to improper training. A training method that could train based on feedback from the environment would be more suitable in these instances.

A related issue is that backpropagation learning occurs entirely off-line. That is, multi-layered Perceptrons generally exist in one of two phases – the learning mode, where correct outputs are provided so that error can be calculated and the weights updated, and feedforward mode, where data arrives at the input layer and processed information is recovered at the output layer. While training, the network does not produce usable output – since we already know the output we want, time is better spent teaching the network until its performance is satisfactory. While running, however, the weights of the network are constant. If the environment changes unexpectedly, or the supervisor's training set was incomplete, the network cannot compensate.

In addition, a multi-layer Perceptron also has no memory other than weights – there is no way to "remember" a previous activation level, which would provide the network the ability to learn temporal patterns. It was noted in the previous section that such temporal patterns and predictions are thought to be key to the brain's ability to learn. A model that incorporates some type of time-

[12] (Jones 2008) 285

based feedback would likely fare better in the face of rapidly changing input (as would be the case, for example, when driving or in a video game).

To solve these issues, we will make two modifications to the standard multi-layer Perceptron. First, we will use TD(λ), a Reinforcement Learning technique that teaches a neural network to learn to maximize the reward from a given environment.[13] Second, we will introduce temporally recurrent connections, which provide the network a way of "remembering" information from previous states. Both modifications have direct parallels to the observed structure and activity of the brain. They will be examined in more detail in the following sections.

2.1.3 Reinforcement Learning

Learning algorithms in artificial intelligence generally fall into one of two categories – supervised and unsupervised learning. Supervised learning involves an expert supervisor who tells the network the correct output for a certain subset of the inputs. The backpropagation algorithm for multi-layer Perceptrons is the most popular supervised learning technique for neural networks. Unsupervised learning, on the other hand, relies on no expert advice. Algorithms for unsupervised learning, such as the Self-Organizing Map,[14] are capable of recognizing patterns in data. Reinforcement learning falls somewhere between these two extremes. While it does not require an external expert to provide it with the correct answer, it does rely on the environment, or a separate entity that senses the environment, to provide it with feedback indicating the relative desirability of the current state of the environment. Of course, relative desirability itself must be based on some aspect of the environment – an aspect that a human supervisor (who has specific ideas of what she wants the artificial intelligence to learn) will have to identify. On the other hand, since this feedback mechanism can often be integrated into a single agent, and since the learning process itself proceeds

[13] (Sutton and Barto, Reinforcement Learning: An Introduction 1998)
[14] (Kohonen 1982)

without direct intervention by a supervisor, reinforcement learning can be considered an unsupervised approach to learning.

The idea behind reinforcement learning is to create an agent that "wants" to learn. That is, reinforcement learning is goal-directed learning based on interaction with an environment. Reinforcement learning is fundamentally evaluative – given an action, the agent evaluates it and predicts the eventual, long-term reward that will come of taking that action. Training is based on estimating the final reward using these intermediary rewards during interaction with the environment.

At every time step (every time the agent evaluates the environment and decides to take an action), it evaluates and selects an action based on how highly the agent predicts it will be rewarded in the long term by taking each action. This selection may be "greedy" – the agent will always take the best option – but generally, especially at the beginning when its evaluation function is still inaccurate and untrained, the agent should explore the state space more to discover more about the environment. For example, the agent may select the greedy option most often, but occasionally select an action randomly (this method is known as the ε-greedy method – the probability of random action is given as ε). When it takes the action, it receives a series of reward signals as (or after, if the action is instantaneous) it carries out the action. It then modifies its reward function – the part of the agent that estimates the reward for taking a certain action from a certain state – so that the value it would produce in a similar future situation is closer to the value it actually received the time it tried the action. In this way, over time the agent will learn to more accurately predict the consequences of its actions. As it does so, it will learn to choose the best action more and more often.

A simple example of the temporal difference method described above is

$$V(s) \leftarrow V(s) + \alpha * [V(s^{'}) - V(s)]$$

V(s) indicates the reward estimate for the current state, V(s') indicates the reward estimate for the next state, and α indicates the fraction of the difference the agent will learn at once, which is reduced over time as the agent narrows in on the best way to evaluate the state of the environment (this is analogous to the learning rate in backpropagation). This simple model of temporal difference operates on a table of possible states. Mathematically, the agent would eventually converge on the optimal evaluation function if it explored enough of the state space.[15] What's more, if the learning rate parameter remains slightly above zero, the agent will continue to learn even if the environment gives inconsistent rewards. This ability to learn accurately, even in the face of an inconsistent environment, makes reinforcement learning in general, and temporal difference methods specifically, highly useful for real-time interactions with realistic environments.

2.1.4 TD(λ)

In the early 1990s, while reinforcement learning was still in its early stages, Gerald Tesauro developed a program known as TD-Gammon to play the game of Backgammon. Using a multi-layer Perceptron neural network and temporal difference methods, he built a program that learned to surpass all other computer programs at the time entirely through self-play – that is, for every move, the neural network would evaluate and suggest a move for both sides and update its predictions after every move it made. Using a temporal difference method known as TD(λ), he had the network learn by playing the game against itself. When using a raw board encoding that imparted absolutely no foreknowledge of the game, the neural network was observed to develop patterns in its hidden layers' weights that suggested that the network learned to discover features of the game entirely without external help.[16] In addition, this network played at approximately an equal playing strength with the author's previous backgammon program, Neurogammon, a multi-layer Perceptron trained with backpropagation on a large series of expert backgammon games. In addition, Neurogammon had hand-designed features given as inputs. When the latter was added to TD-Gammon, the program played near the level of human masters.[17]

[15] (Sutton and Barto, Reinforcement Learning: An Introduction 1998) 13
[16] (Tesauro, Temporal Difference Learning and TD-Gammon 1995)
[17] Ibid

Temporal difference learning is a powerful concept. It is capable of learning directly from raw experience with the environment (that is, without a model of how the environment works), and it can do so by learning from its own previous estimates without waiting for a final result.[18] The TD(λ) algorithm is an alternative weight update scheme for multi-layer Perceptrons which uses temporal difference methods to calculate the weight change rather than the standard delta rule. As in the delta rule, the error – in this case, the difference between two successive predictions – is back-propagated along the dendrites of each neuron. At each neuron, the network gradient is computed – this is the same gradient used in the delta rule shown previously. With temporal difference, the weight (*w(i)* at time t+1) is updated by the following formula:

$$w(i)_{t+1} = w(i)_t + \alpha * \big(r(t+1) + Y(t+1) - Y(t)\big) * e(t)$$

As before, α is the learning rate. Y(t) is the estimated reward at time t, and r(t) is the actual, received reward at time t. The last term is known as the *eligibility trace*. The eligibility trace essentially determines which and how much previous states will affect the current weight change. Before updating the weight, each dendrite updates its eligibility trace by first decaying the previous value and then adding in the current network gradient times the error:

$$e_t = \gamma\lambda e_{t-1} + \boldsymbol{g} * Y(t)$$

The γ parameter is the discount rate – how much the reward from the current state should affect previous states.[19] In Tesauro's network, this was set to 1.0 as the network would never experience reward until the end of the game (in which case you would want every state to equally benefit/suffer from the reward). The λ parameter is the trace-decay parameter, which is used to determine how much credit should be assigned to previous states for the current temporal difference error. The difference between the two parameters is subtle – γ focuses on limiting the effect of a single reward to a certain number of states, but λ focuses on doing the same for the difference

[18] (Sutton and Barto, Reinforcement Learning: An Introduction 1998) 134
[19] Ibid, 58

between two successive predictions. In Tesauro's TD-Gammon, since only one actual reward is ever observed, γ should be one since the reward is zero until the end of the game. The game does make predictions about its chances at each move, though, and the difference between two predictions (that is, the temporal difference) should only affect a limited number of states.

The combination of TD(λ) with neural networks thus provides a method to solve what is known as the "temporal credit assignment" problem. With many problems, we know intuitively that it is a series of actions that lead to an eventual reward. Traditional backpropagation can take the final time-step and use the difference between the final prediction and the actual reward as the basis for learning. While the network would become better at predicting the final reward from the final state, however, no changes based on that reward will specifically affect previous states. TD(λ)'s eligibility traces solve exactly this problem – how to determine the effect of previous states on the final reward.

Unfortunately, despite their wide availability since 1988, TD(λ) methods with neural networks have had little use in the field of AI due to a major theoretical limitation – while TD(λ) is proven to converge to the correct solution when used with linear function approximators, there is no guarantee that TD(λ) will converge to the correct solution – that is, the function approximator's solution will not improve towards the correct solution, even given infinite training time and infinitely many training samples – when used with nonlinear function approximators, such as neural networks.[20,21] Despite this major theoretical shortcoming, however, impressive results – such as those from the TD-Gammon project mentioned previously – have been obtained using TD(λ) with neural networks. It is therefore hoped that positive results can be achieved, especially when combined with the genetic algorithm described below.

[20] (Hamid R. Maei 2009)
[21] As of early 2010, a new algorithm called GQ(λ) has been created which follows from TD(λ) but is proven to converge for nonlinear function approximators: (Maei and Sutton 2010)

Temporal difference methods are very similar to well-known neurological reinforcement functions. The chemical neurotransmitter called dopamine is closely related to reward. It has been specifically shown that dopamine is used by the brain to signal an error in the prediction of reward.[22] The method of this prediction is stunningly similar to the temporal difference methods described above – in fact, temporal difference methods have been used to interpret the activity of dopamine neurons used in reward-dependent learning.[23] These data suggest that reinforcement learning, especially temporal difference methods, have a strong hand in helping complicated intelligent creatures – humans and other animals – to learn based on reward. Applying this idea to an artificial neural network brings us closer to a biologically plausible model of intelligence that works effectively on computers.

Unfortunately, integrating the temporal difference theory into neural network theory is biologically problematic. While temporal difference is a biologically sound way to assign credit to states based on future rewards (i.e., it solves the *temporal credit assignment* problem), we must combine it with our existing learning method, backpropagation (which solves the *structural credit assignment* problem), in order to completely integrate it with the multi-layer Perceptron model. With regards to backpropagation, there has been some debate as to whether signals are only started at axons, or whether signals can also be initiated at dendrites (as in backpropagation). A recent study suggests that that while signals are only initiated at axons, in some neurons the activation also propagate backwards into the dendritic tree.[24] The authors suggest that this backpropagation may have a role in synaptic plasticity, which makes artificial backpropagation at least biologically plausible. With regards to the calculation of the network gradient in backpropagation, however, the case for biological plausibility is very weak, since it requires that the error signal go through the derivative of the activation function. In neurons, this activation function is a step function, which is not differentiable.[25]

[22] (Pöppel and Arrias-Carrión, 2007)
[23] (Dayan, Montague and Schultz 2007)
[24] (Häusser, et al. 1997)
[25] (de Castro 2006) 173

2.1.5 Temporally Recurrent Connections

While many aspects of neurobiology have found their way into various types of artificial neural networks, one notably scarce variation is the presence of feedback connections. There exist network models which allow for recurrent and feedback connections, but generally they are evaluated statically – a feedback loop is identified and iterated either a fixed number of times or until it reaches a steady state. While this allows traditional training algorithms, such as backpropagation, to work, it is biologically implausible for neural processing to halt while a feedback loop is resolved, especially given the number of such loops in the brain. In 1990, Jeff Elman introduced one of the first implementations of a recurrent network based on time delay. In his simple three-layer network known as a Simple Recurrent Network, the hidden layer neurons fed back into neurons in the input layer, allowing them to recall the previous state. While the recurrent connections had a weight, the recurrent weights were set as constant.[26] Further work on recurrent networks has led to few major advances. Modern recurrent networks are generally trained with genetic algorithms or similar approaches, resulting in very slow training times.

A more biologically plausible model may be to treat recurrent connections similarly to other connections. In a standard feedforward multi-layer Perceptron, recurrent connections can be introduced in the following manner: the first (input) layer receives input as usual. All neurons connected to those neurons then receive input. The second layer neurons now have input from all previous layers, so those neurons fire. The third layer will now also have input from all preceding layers, and its neurons will fire. Any time a neuron fires, it resets its activation level. When a neuron in a later layer connects to a neuron in a prior layer, the receiving neuron will receive a raised activation level. When the network is run subsequently, any neurons that received feedback connections will start with elevated activation levels. In this way, normal feedforward activation is not interrupted, but state information from previous runs is maintained.

[26] (Elman 1990)

Training works in a similar way. When propagating errors backwards, the error activation is set to zero in a neuron. As with feedforward operation, a layer of neurons is activated when all later layers have been activated. Neurons with feedback connections will receive error after they have back-propagated their error signal. Therefore, the next time the network is trained, the neuron with the feedback connection will have an elevated error activation level. This has the effect of delaying by one cycle the effect of that connection's weight on the error. Mathematically, the numerical value of the error is the same; its effect is simply delayed by one time step.

This model of network, wherein neurons can "pre-activate" other neurons in advance of them receiving input, is better supported by current neuroscience. Unlike temporal difference learning, which defines how to assign credit to previous states (that is, it answers the question *how much did state (t) contribute to the reward received at state (t+x)*?), temporally recurrent connections define how to present information gleaned from previous states to future states – essentially, they include information from previous states in the current state. This allows the network to remember information from previous states, which may allow it to react more consistently to environmental factors that change predictably through time.

2.2 Genetic Algorithms

Unlike artificial neural networks, which must make vast simplifications on the model of the brain (which itself is poorly understood), the idea behind evolution is fundamentally simple. In nature, natural selection favors the fittest – those creatures that have the best capability to survive are more likely to do so and will thus be more likely to pass on the genes that define their traits through reproduction. Over billions of years, nature has expanded from primordial slime to today's humans and other animals that exhibit intelligence. The power of evolution comes from the astounding variety present in genetics. The genetic code that makes up a living creature is made up of sequences of DNA known as genes. These genes encode the creation of the amino acids and proteins that make modern, complex animals possible.

In 1975, John Holland introduced the modern version of the genetic algorithm. Using solutions encoded as "chromosomes" where each gene was a bit, he introduced the genetic operators of crossover (where the child takes some genes from each parent) and mutation (where a few genes from the resulting child are changed randomly) to maintain diversity in the population even while allowing only the most fit to breed.[27] Modern genetic algorithms need not deviate significantly from this original formula – while modern computers allow for faster operation and larger chromosomes which may represent more complex objects (such as artificial neural networks), the general algorithms used today are still largely identical to Holland's.

Before starting the algorithm, a genomic representation of what is being evolved is created. This representation must lend itself naturally to modification by crossover and mutation; if not, the algorithm may generate invalid genomes which must be discarded. Then a population of random genomes is created. A larger population allows for more diversity, but can significantly slow the algorithm down depending on the method for fitness evaluation. Then each genome is evaluated to test its fitness. This process may be time-consuming – the genome (the genotype) must be converted into an object (the phenotype) and the object must be tested in some way (for example, a neural network agent may be tested in its target environment). Then a set of genomes is selected for breeding. The most common method of selection is "fitness-proportionate" selection. This method of selection assigns each genome a portion of a roulette wheel according to its fitness – those with the highest fitness have the most space on the wheel. Then the wheel is spun a number of times, producing the selected genomes. It is common for the same genome to be selected multiple times; however, if a single genome is much, much better than others, diversity will suffer because the same genome may make up most of the breeding sample. In that case, a simpler method such as tournament selection (where a single tournament involves a small number of genomes selected randomly from which the fittest is chosen) may be useful.

[27] (Jones 2008) 197

After selection, two genomes breed using the genetic operators, crossover and mutation. In crossover, a child receives half of its genes from each parent. Normally one- or two-point crossover is chosen – at each "point" the child switches which parent it gets its genes from; however, a "uniform" crossover may also be used, where the child chooses each individual gene from one of its parents randomly. Both children produced by crossover will have genes from the opposite parent in each position. In mutation, each gene in each child genome has a small chance of being modified. Now the population contains all of the children, as well as the best parents from the previous generation (including parents ensures that the algorithm doesn't throw away a known-good solution).

In its most basic sense, a genetic algorithm is essentially a search algorithm. At first, a population of genomes is randomly created. As the algorithm progresses, the members of the population explore the space of possible solutions through mutation and crossover, creating new solutions and improving the overall fitness. Even as the genetic algorithm produces better and better solutions to the problem, it avoids local maxima by constantly introducing changes in the form of mutations and even new combinations created by crossovers. When the state space is extremely large, and especially when the environment gives little or no direction on how to develop a good solution, a genetic algorithm can often find a solution out of nothing but a random starting population.

2.2.1 Evolving Neural Networks

Significant work has been done on evolving various aspects of neural networks using a genetic algorithm. Many difficult problems in neural networks can be solved in this manner, such as the selection of connection weights, the selection of the neural network architecture, and the selection of features to provide to input. Weight evolution is generally used only when supervised options are insufficient (that is, for training sets where the problem has no known answer or where the target environment does not provide an easy measure of reward for use with reinforcement learning). One example of successful weight evolution is the Blondie24 checkers AI project, which did not change the neural network architecture but used a genetic algorithm to evolve the weights over time. After fewer than one thousand generations, the neural network was able to play at an expert level of checkers.[28] While a genetic algorithm for evolving weights can be effective, however, it is not biologically satisfactory – while genetics does determine the structure and capabilities of the brain, the actual learning that takes place does not appear to occur in an evolutionary manner.

Selecting a network's architecture through a genetic algorithm is both biologically plausible and technically effective. Without evolutionary techniques, a human expert will generally engage in a trial-and-error process to create an architecture that will have enough nodes to solve the problem but not so many that the network trains slowly.[29] A genetic algorithm can create tightly pruned but effective network architectures, even for recurrent neural networks.[30] In a similar manner, genetic algorithms have been used to reduce the set of inputs. This is especially useful in large problems where there may be hundreds of possible inputs. After optimizing the set of inputs with a genetic algorithm, studies have noted reduced training time and smaller errors with fewer input features.[31] Both of these problems occur in the natural brain. The structure of the human brain, for example, is significantly different even when compared to other animals. The anterior (front) half of the neocortex, for example, is much more heavily connected to muscle movement in humans than in other animals, giving rise to more complex movements and behaviors.[32]

Most genetic algorithms use chromosomes that are as concise as possible. The human genome, however, is highly redundant, with both unused sections of code (that is, DNA that is part of an "intron," a sequence of genes that are not expressed) and many different groups of genes may code for the same thing. As an example of the latter, each sequence of three DNA molecules (called nucleotides) represents a single amino acid, of which there are twenty. In addition, there are four types of nucleotides (adenine, guanine, cytosine, and thymine). Three nucleotides therefore allow for up to sixty-four possibilities – far more than the necessary twenty. It turns out, however, that each

[28] (Fogel 2002) 273, 284
[29] (Yao 1999)
[30] Ibid
[31] Ibid
[32] (Hawkins 2004) 103

amino acid can be coded for by several (between one and four) different arrangements of nucleotides. The redundant codes are known as degenerate code.

While many evolutionary experiments have been carried out without notice to this particular piece of biology, a few studies have noted significantly increased performance when introns are added. One study, which specifically evolved neural networks, tested the usefulness of non-coding regions in the chromosomes used.[33] As an initial experiment, a population of neural network architectures for binary counting was evolved, using a genetic code containing both coding and non-coding regions. For this first experiment, the introns were removed after each generation. This resulted in almost no evolution. When introns were left alone, however, evolution produced significantly better networks, suggesting that accumulated changes in non-coding regions stored potentially useful information that could be expressed easily with just a few mutations

[33] (Burroughs 2005)

2.3 Pac-Man

Originally released in Japan as "Puck Man," *Pac-Man* became a massive hit, selling over 300,000 arcade machines worldwide (100,000 in North America compared to just 70,000 for the second-place *Asteroids*).[34] The game places the player in control of Pac-Man, an anonymous, circular yellow mouth whose only goal is to eat dots and earn points while avoiding the four ghosts whose greatest desire is to hunt down and eat Pac-Man. If eaten, Pac-Man loses one of its lives (generally the player starts out with three lives, but this is actually configurable in the arcade cabinet to be one, two, three, or five). The game is played in a simple maze filled with 244 dots – 240 regular dots, which provide Pac-Man with ten points each, and four large flashing dots known as "energizers" which, when consumed for fifty points, temporarily allow Pac-Man to eat the ghosts for extra points (the ghosts become "frightened") – 200 for the first, and doubling for each additional ghost (for a total of 1600+800+400+200 = 3000 possible extra points per energizer) until either time runs out and the ghosts begin chasing Pac-Man again or Pac-Man eats another energizer, thus starting the cycle again. When a ghost is eaten, it returns to its starting location to begin hunting Pac-Man again. Finally, twice during each level (after 70 dots have been eaten and after 170 dots have been eaten) a fruit appears in the center of the maze for approximately 10 seconds, which is worth from 100 to 5,000 points depending on the current level.

After Pac-Man consumes all 244 dots, a new level begins, where the speed is increased, the type of fruit changes, and the amount of time that Pac-Man has to eat the ghosts after eating an energizer (known colloquially as "blue time" since the ghosts turn blue) is reduced. Eventually, ghosts cease to turn blue entirely and their normal speed becomes faster than Pac-Man's, making the later levels extremely challenging.

The game also includes a number of complications to aid the player, especially once the ghosts begin moving faster than the player. First, the artificial intelligence guiding the ghosts forbids

[34] (The Dot Eater 2006)

them from reversing direction except in certain, well-defined circumstances. Second, several times during the level the ghosts' artificial intelligence reverts to "scatter" mode, where the ghosts stop chasing Pac-Man and briefly head to their home corners (the top right for the red ghost (Blinky), the top left for the pink ghost (Pinky), the bottom right for the blue ghost (Inky) and the bottom left for the orange ghost (Clyde). This is one of two instances where the ghosts are forced to turn around. Third, Pac-Man can eat an energizer, which allows him to eat the ghosts in the early levels. Even if the ghost does not become frightened, it will still turn around, potentially providing Pac-Man a chance to escape. Fourth, a tunnel allows the characters to travel off one side of the screen and reappear on the other. While Pac-Man can travel normally through the tunnel, the ghosts slow to almost half their normal speed, allowing Pac-Man to easily escape close pursuers. Fifth, two T-shaped areas of the maze, both directly above and below the monster pen, are protected from one end – unless frightened, ghosts are unable to turn up into either of these areas, allowing Pac-Man to completely throw off close pursuers. In fact, if the player is smart, she can place Pac-Man in one of these T-shaped structures and completely avoid the ghosts indefinitely, although it will be impossible to eat more dots. Sixth, and finally, when rounding corners, ghosts will travel to the middle of an intersection and make a 90-degree turn. When Pac-Man enters an intersection, however, he is able to move diagonally into the turn, essentially doubling his speed for a crucial split-second.[35]

Soon after the game's release, many players noticed that, by memorizing certain patterns, they could proceed past any level without harm. These patterns, when executed perfectly, always played out in the exact same way. Later analysis of the assembly source code of the game revealed that the game was entirely deterministic – the artificial intelligence guiding the ghosts was based entirely on conditions in the maze (usually involving the location of Pac-Man). When frightened, the ghosts choose a direction based on a pseudo random number generator (PRNG). The generator itself is exceedingly simple: it simply reads the last two bits from a semi-random location in the game's RAM. Unfortunately, the game resets the "seed" – the number that determines where in the

[35] (Pittman 2009)

sequence of random addresses the generator is – every time a new level begins or Pac-Man dies. Therefore, in the context of a memorized pattern, the ghosts will behave in the exact same way when frightened.[36]

After the machine code for the game was disassembled and analyzed, enthusiasts familiar with the assembly language for the Zilog Z80 processor used in the arcade machine were able to discover many of the game's closely held secrets, including the ghost artificial intelligence routines.[37] It was discovered that the artificial intelligence routines are surprisingly simple for the complex behavior they create. Basically, when it reaches an intersection, a ghost uses a particular routine (unique to each color ghost) to determine a "target" tile in the maze (a tile is an 8 by 8 pixel square, which the game uses to delineate individual sections of the maze; to determine if two objects are touching, the game simply has to ask whether they are present in the same 8x8 square). Then it takes the turn that reduces the Euclidean distance between its current tile and its target tile the most. Ties are broken simply – the ghosts prefer directions in the order up-left-down-right. The red ghost's tile is simply Pac-Man's own tile, making him difficult to shake off. The pink ghost aims four tiles in front of Pac-Man (along the direction he is facing, although a bug causes the tile to be four tiles up and to the left if Pac-Man is facing up[38]), allowing him to help Blinky by getting in front of Pac-Man and trapping him. The blue ghost is the most complicated – to visualize his target, one should draw a line segment from the red ghost's position to the tile two spaces in front of Pac-Man (using the same faulty logic for "up" as Pinky), and then double its length. The end of this line segment is his target. This keeps him far from Pac-Man if Blinky is far away, but as the red ghost closes, the blue ghost comes in to help. The least dangerous ghost is Clyde, the orange ghost. When its Euclidean distance from Pac-Man is fewer than eight tiles, it heads for its scatter target – the lower left corner. When it is further away, it heads directly for Pac-Man like the red ghost. While the orange ghost is skittish, it

[36] Ibid

[37] There are a relatively large number of such enthusiasts – the Z80 processor was and still is one of the most popular 8-bit processors, seeing use on many home computers in the late 1970s, the Nintendo Game Boy in the 1980s and 90s, and the popular Texas Instruments line of graphing calculators, including the TI-73, TI-81, TI-82, TI-83, TI-84, TI-85, and TI-86 models, the 73, 83, and 84 versions of which are still sold as of 2010.

[38] (Hodges 2008)

makes the bottom left corner relatively dangerous for Pac-Man, making that area an early target for experts.

The game has several properties that make it interesting for artificial intelligence work, especially with reinforcement learning. Firstly, the game lends itself to a very simple quantitative evaluation of "reward;" that is, a better player will score more points and live longer than a worse player. Secondly, the artificial intelligence can be evaluated (by a human) based on several qualitative measures. Pac-Man as a game requires extensive task prioritization and risk assessment. Essentially, a good player balances four major priorities – avoid the ghosts, eat dots, eat energizers, and eat frightened ghosts. A simple player may excel at one or two of these categories; for example, the player may be good at eating dots and energizers but unable to effectively avoid the ghosts. A more advanced player will exhibit more nuanced strategies involving relationships between the different priorities. For example, a player may wait near an energizer until many of the ghosts are near before eating it, allowing easier capture of the frightened ghosts. Finally, the game, as mentioned before, is entirely deterministic. If a neural network plays the game many times without modifying the weights, the results are identical, removing any doubt about a computer player "getting lucky" and erasing uncertainty in measuring reward.

Some work has shown promise in developing artificial intelligence agents to play as Pac-Man. One recent study, for example, evolves the weights of neural networks using evolution strategies (a version of the genetic algorithm designed to work with real-valued parameters, like the weights of a neural network). The study uses minimally processed input – data from only a small area around Pac-Man was used.[39] Processed information, such as the distance between Pac-Man and the various ghosts, is not included, though the number of dots in each direction from the player was used to help prevent the player from getting stuck. Using a standard feedforward network with 79 inputs and a single hidden layer with eight neurons, the resulting evolved program was able to clear half of the

[39] (Gallagher and Ledwich 2007)

dots in a maze with 236 total dots. For such a small network – just eight hidden neurons – this result is fairly impressive. The version of Pac-Man used, however, is highly simplified – only one ghost is used, the energizers are removed, and the fruits are not present. In three trials of the evolutionary experiment, two of the trials resulted in a high score over 1200 (over 120 dots found). Increasing the complexity to non-deterministic ghosts (the ghost follows Pac-Man 90% of the time, and 10% of the time picks a random direction) and adding more ghosts, however, indicated that the model was not able to scale to more difficult problems very well – the best trials were able to eat only 50 or so dots before being captured.

This work provides both a sobering example – none of the models were able to complete even a single maze, even in a highly simplified environment – and a reason for hope. The model the authors present has room for improvement in several areas. First, the neural network input is only the area local to Pac-Man – no information is provided about any other area of the maze except to say how many dots lie in each direction from Pac-Man. This means that until the ghost is very close to Pac-Man, the artificial intelligence has no idea of its existence! A closely related problem is the fact that this inherently temporal problem, where the direction of both ghosts and Pac-Man himself can have an effect on the game, no information is provided about them (nor can the network deduce them by observing and remembering a series of positions). In addition, the number of hidden units is very small – given the size and complexity of the input vector, a much larger layer of hidden nodes would likely be required.

Another study used a similar approach as above, but used a preprocessed set of input features (such as the distance to the nearest dot and the distances to the ghosts) to guide a simple perceptron, and rather than using four outputs to choose a direction, the author's program tested locations near the player and chose the location that was evaluated as the best.[40] The author used a version of Ms. Pac-Man with extremely simple random ghost AIs – they simply followed Pac-Man

[40] (Lucas 2005)

for some percentage of the time, and otherwise moved randomly. The results of the paper were fairly encouraging – at least one network was able to complete an entire level by eating all of the dots.

The learning method used in both of the above projects (evolving weights with a genetic algorithm) can be useful in some circumstances, but it does not cater itself specifically to the problem at hand. The dynamic nature of a video game is better suited to real-time learning, which can not only take advantage of individual interactions in the environment, but it is more "granular" – rather than punishing a solution which performed badly overall (but had one or two moments where it did something brilliant), individual actions can be punished or rewarded depending on how well they perform in the long run. The ideal method of learning for this type of problem, of course, is reinforcement learning.

One other very recent study, by the same authors as the Ms. Pac-Man study just discussed, provides initial results comparing a genetic algorithm and a neural network trained with temporal difference learning.[41] This version of the game uses the same ghosts as before but provides the network with two inputs – the distance to the nearest "escape node" (the closest tile where there is not a ghost in all four directions) and the distance to the nearest dot or energizer. The authors noted very poor performance for the temporal difference neural network, and small gains with the genetic algorithm, though both solutions showed results that were only modestly different after training compared with before training. A major issue, which the authors note in their conclusion, is that the input specification for the escape node is extremely noisy – a minor change in the board configuration can result in a large change to the input given to the neural network. The authors note that their previous work involving temporal difference learning showed that noisy inputs had a significant impact on the final ability of the network to converge; therefore, a major point to take away from this particular study is to choose a set of inputs to the network that both provides valuable information and changes predictably over time.

[41] (Burrow and Lucas 2009)

3. Methods

3.1 Neural Network Model

ArtificialNeuralNetwork.py implements an object-oriented library for building neural networks with an arbitrary architecture in the Python programming language.[42] By default, when building a network with multiple layers, the input layer automatically has a hidden bias neuron created. The bias neuron always emits an impulse of 1.0, and is fully connected to the next layer. The program allows for training using both traditional backpropagation and TD(λ). The starting network for all experiments was a simple feedforward network with two hidden layers. Initial weights were distributed uniformly randomly in the range (-0.1, 0.1). The primary limitation of this implementation is speed – networks with thousands of connections may take several milliseconds to run, making them impractical for use in real-time games. To mitigate this problem, users with the 32-bit version of Python on x86 computers can use Psyco,[43] a specialized just-in-time compiler which approximately doubles the execution speed of large networks.

3.2 Genetic Algorithm Model

GeneticAlgorithm.py implements a standard Genetic Algorithm as described previously. The algorithm operates on a population of Genomes, which each have a standard set of Chromosomes. Each chromosome has a type – binary, real (for floating point values), and combinatoric (for representing problems like the travelling salesman problem). For evolving neural networks, two problems must be solved – evolution of weights, and evolution of network structure. To effectively evolve both, a slight modification of the chromosome structure from (Burroughs, 2005)[44] is used.

[42] http://www.python.org
[43] http://pscyo.sourceforge.net
[44] (Burroughs 2005) 8

The chromosome is broken into 64-bit blocks. As in Burroughs's paper, we encode both weights and connections in a single block. In this implementation, the first 32 bits of a block encode a 32-bit IEEE-754 floating point number. The second 32 bits have two purposes. The three least significant bytes encode the node number from which this node receives a connection, with enough upper bits dropped so that the number is less than the total number of nodes. Special values in the most significant byte encode the beginnings and ends of nodes and layers – 0xFA and 0xFB to start a node and 0xFC and 0xFD to end a node, and 0xFE to begin a layer and 0xFF to end a layer.[45] As in the previous work, anything appearing between a node or layer end marker and the next node or layer start marker is discarded during network creation (though this "junk" DNA does remain in the genome and can be reactivated during subsequent evolution). While a "start layer", "end layer", or "start node" special value can, coincidentally, appear between a start node block and an end node block, these values have no special effect unless they are applicable ("start layer" is interpreted only after "end layer", "end layer" is only interpreted after "end node" or "start layer", "start node" is only interpreted after "start layer" or "end node", and "end node" is only interpreted after "start node").

Two special values appear at the start of the network. The first value is interpreted as the number of outputs. Since this is fixed for the problem, this value is considered immutable; mutation will not attempt to corrupt it. The inputs are determined in a separate chromosome.

The previous paper did not find it necessary to encode layers directly since feedback loops were resolved as independent units. Temporally recurrent connections, however, require that nodes with feedback connections to previous layers be computed after those layers have already been computed (so that the recurrent connection provides information to be used in the next state, not the current state). Thus, for simplicity, layers are directly encoded in the chromosome. Otherwise, networks are constructed as in the previous paper – a series of taint sweeps are performed to remove

[45] 0x is the common prefix for hexadecimal (base 16) numbers, in which each number represents four bits. 0xFA through 0xFF are hexadecimal for the decimal numbers 250 through 255.

disconnected nodes (those nodes that do not connect to the input or contribute to the output). Then the layers and connections are constructed according to the specification.

Each Genome also has an additional chromosome which supplies additional parameters for the neural network. This chromosome is made up of six 32-bit blocks. The first block represents the activation function (linear, logistic, or hyperbolic tangent) for the hidden layer as follows (values in hexadecimal):

- 0000 to 1FFF represents a linear ($f(x) = x$) activation
- 2000 to 8FFF represents a logistic ($f(x) = 1 / (1 + e^{-x})$) activation
- 9000 to FFFF represents an hyperbolic tangent ($f(x) = 1.7159 \times tanh(\frac{2}{3}x)$) activation[46]

The numbers were chosen such that logistic and hyperbolic tangent functions would have a much higher probability (7/16 each) of appearing than the linear activation (1/8). This distribution was chosen because neural networks which only use linear activations in their hidden layers are only capable of linear function approximation, and while it's possible that a linear solution to Pac-Man will do better than a nonlinear solution, it seems unlikely given the complicated nature of the game (on the other hand, a linear network may also allow temporal difference learning to progress more rapidly).

The second 32-bit block encodes the activation function for the output layer. In this case, each activation function has an equal probability:

- 0000 to 5FFF represents a linear activation
- 6000 to AFFF represents a logistic activation
- B000 to FFFF represents an hyperbolic tangent activation

[46] The values for the tanh() equation are from (Haykin 2009) 146

The third 32-bit block represents the value for the γ parameter in temporal difference learning as an IEEE-754 floating point number. This value starts out as uniformly randomly distributed between 0.8 and 1.0. The fourth 32-bit block represents the value for the λ parameter in temporal difference learning. This value starts out as uniformly randomly distributed between 0.0 and 0.8. These starting values represent their relative priorities – we always want the discount rate (γ) to be fairly close to 1 so that the agent is "farsighted;" that is, it learns to make actions that maximize future rewards. On the other hand, experiments have indicated that small to intermediate values of λ tend to produce the best results.[47,48] The fifth 32-bit block represents the value of the learning rate (α) parameter, which starts out uniformly and randomly distributed between 0.01 and 0.15.

The final 32-bit block contains the input specification. These inputs will be explained in more detail later. Each set of three bits codes for a single input – if any of the three are 1, the input will be used; otherwise the neural network will not receive the given input. Bits 31 and 32 (the two most significant bits) have no effect.

Bits 1 – 3	Bits 4 – 6	Bits 7-9	Bits 10-12	Bits 13 – 15
Ghost dist.	Frightened ghost dist.	Ghost speed	Ghost direction	Player position
Bits 16 – 18	Bits 19 – 21	Bits 22 – 24	Bits 25 – 27	Bits 28 – 30
Nearest dot	Nearest energizer	Dot ratio	Nearest junction	Nearest Fruit

Child networks were created by crossing-over two parent networks and then applying mutation. Both operate on entire blocks (64 bits in the first chromosome, and 32 bits in the second). Each crossover operation produces two child networks, each containing information from opposite parents. For the first chromosome, the networks are effectively placed on top of each other and scaled so that each unique position in the longer chromosome equates to a not-necessarily-unique location in the shorter one, and where the starts and ends are aligned:

[47] (Sutton, Learning to Predict by the Methods of Temporal Differences 1988) 21
[48] (Tesauro, Practical Issues in Temporal Difference Learning 1992) 270

A_1	A_2	A_3	A_4	A_5	A_6
B_1	B_1	B_2	B_2	B_3	B_4

An example alignment - a child will consist of either A1|B1|A3|B2|A5|B4 or B1|A3|B2|A5|B4.

Then a segment size is randomly chosen by the equation $20 + (750 - 20) * x^3$, where x is a random number between 0 and 1. This sets the range between 20 and 750, with a bias towards slightly shorter segments due to the exponent on x.[49] After a segment is chosen from one parent, the next segment starts in the equivalent location (given by the alignment) in the other parent. Then a child is constructed by taking segments alternately between each parent. The starting parent is chosen randomly. To increase the diversity of the resulting population, only one child is created between each pair of parents.

After child networks are created through crossover, they are mutated. The mutation operator has four forms – insertion, deletion, duplication, and "point." The chromosome was scanned linearly until a block was selected randomly for mutation. When a block was selected, the mutation type was chosen at random (however since the size of the second chromosome is fixed, and the position of the node determines how it is interpreted, it is only mutated through point mutation). Due to the nature of IEEE-754 floating point numbers, where changing one bit can have the effect of multiplying the weight by up to 2^{127} (the maximum size of the exponent for 32-bit floating point numbers), mutation operates specially on the floating-point values. These values are instead uniformly randomly chosen in the range (-0.1, 0.1). When float values are mutated, they are simply increased or decreased by a uniformly distributed randomly generated number between -0.1 and 0.1. The second half of the block is mutated by generating a new block of random bits. Insertion creates a random block and inserts it before the block selected for mutation. Deletion removes a block. Duplication takes a series of blocks (uniformly randomly chosen from the range [2, 20]) and copies it to another location. The location chosen is random, but normally distributed with a mean of 1000

[49] (Burroughs 2005) 18

blocks and a standard deviation of 50 blocks. Point mutation replaces an existing block with a randomly generated block. The mutation rate used was 2%.

Occasionally, mutation or crossover will produce a network which either has some disconnected outputs, some disconnected inputs, or nodes with the same or more incoming feedback connections than feedforward connections, which could lead to an infinite loop. In any of these cases, the genome is considered "non-viable." In the case of disconnected inputs or outputs, the error is detected during crossover. In this case, one of the parents is randomly selected and mutated using a doubled mutation rate. If a neuron with too many feedback connections is detected during run-time, the genome is given a fitness of 0.0 and will have no chance of contributing to the future population. Tests indicated that while disconnected inputs or outputs could be relatively common (occurring more than 10% of the time), networks with feedback loops were extremely rare (occurring less than 0.1% of the time).

The genetic algorithm used fitness-proportionate selection to select the parent genomes that will contribute to the next generation's population, as described in the previous chapter. In addition, the algorithm always saves some of the best genomes from the previous generation. This process of "elitism" ensures that a good solution is not lost through crossover or mutation. Since this "carried-over" genome has already been evaluated, its inclusion does not increase the computational time required for fitness evaluation in subsequent generations; however, it does allow it to continue to breed with other solutions.

3.3 Nom Mon – A Replica of "Pac-Man"

A custom remake of the original Pac-Man arcade game, called Nom Mon, was created for this project. Written in Python, this version of the game retains the most important game-play features of the original – ghost behaviors, maze layout (including the tunnel, the pen, the number and location of dots, and the restrictions on ghost movement in the two T-shaped structures above and below the pen), fruits, energizers, and the relative speeds of Pac-Man and the ghosts. Due to complexities in the way the original arcade game was implemented – it relies on hardware CPU interrupts for some of its timing – events in Nom Mon, such as exactly when the ghosts leave the pen and exactly when the ghosts switch from scatter to chase, do not occur exactly as they do in the original. While the exact patterns that expert players use in the arcade game would not work directly on this new version of Pac-Man, the game retains both the determinism that makes experiments repeatable and the complexity that makes the game interesting, both for AI research and for human play.

When run with a neural network, Nom Mon does not limit the frame rate. It takes between 1 and 2 milliseconds to draw each frame. The neural network itself takes another 0.5 to 2.0 ms when Psyco is used, depending on the size of the network. To maximize the speed of the game, the neural network is allowed to change directions only once every eight frames. In practice, this does not provide a serious disadvantage – it takes from seven to eight frames to move from the edge of one tile into another, and there is always at least one tile between two turns. Thus, the game is able to run at about 250 to 1,000 frames per second, or between four and sixteen times faster than the game at its normal speed. This allows about 5 games to be completed per minute, depending on how well the network plays (for example, the longer the player stays alive, the longer it will take to complete a game).

Nom Mon provides an interface to translate the current game state into a succinct input format for the neural network. While it was initially hoped that a "raw" encoding would be sufficient, the previous Pac-Man experiments demonstrated that the complexity of the input made a large difference in the final ability of the networks. Therefore, we decided to use a simplified set of preprocessed input for the neural network. A previous paper,[50] which taught a neural network to play Pac-Man using Evolution Strategies, uses a set of network inputs that provides a useful starting

[50] (Lucas 2005)

point.[51] In that paper, the author provides inputs telling the shortest path distance to several important features – the nearest dot, the nearest energizer, each of the four ghosts (two inputs each – one for chasing ghosts and one for frightened ghosts), and the nearest junction (a junction is defined as a maze location from which the player can proceed in at least three different directions). The author also provides the current location of the player. If an item doesn't exist, its distance is set to the maximum. Nom Mon scales all values from 0.0 (the item is very far away or not on the map) to 1.0 (the item is situated in the same place as the player character), except for location, which is scaled from (-1.0, -1.0) for the top left and (1.0, 1.0) for the bottom right. This input scheme is designed to convey "urgency" about a particular item – if it is close, the input provides the network with a higher impulse, but as the item gets further away, the impulse weakens.

To that basic set, Nom Mon provides a few additional inputs. The first is simply the shortest path distance to the fruit if it is present on the board (the previous author's implementation did not include fruits). Another added input is the ratio of dots in the direction the player is facing to the total number of dots (expressed as a value between 0.0 and 1.0). This gives the computer a hint as to which direction leads to the greatest number of points. The game also provides an extra input for each ghost telling their speed. This can be useful for teaching the program to use the tunnels and energizers effectively. Speed is scaled so that 1.0 is Pac-Man's speed and 0.0 is stopped. Finally, one extra input is used for each ghost telling whether they are moving towards Pac-Man (1.0) or away from him (0.0).

While the evolution experiment discussed in the previous chapter used multiple output neurons (one for each direction), temporal difference learning is fundamentally an "evaluative" learning method, as stated in the previous chapter. Thus, when attempting to choose a move, the neural network must evaluate the positions resulting from a particular move. Nom Mon predicts the position of the network's player after 8 pixels of movement in each given direction (taking into

[51] (Lucas 2005) 203–210

account the effect of walls) and calculates the variables given in the previous paragraph from this updated position. Then the network must evaluate each of the four positions.

To facilitate exploration of the state space, the program will not always choose the best evaluation of the positions. In the beginning of training, when the predictions are presumably the least accurate, the probability of choosing a random action rather than the best action, ε, is set very high. As the network gains more experience the value of ε should decrease accordingly. Since there is no reliable metric of prediction accuracy for temporal difference methods, we used a schedule for epsilon based on the number of games played: $\varepsilon = \frac{1}{(5\times g)+1}$ where g is the number of games played so far. Thus, the probability of selecting a random action starts at 100% but quickly drops to 10% by game 5 and 1% by game 20.

As the neural network plays the game, it receives a reward on each time step to positively reinforce good actions (such as eating dots and avoiding ghosts) and negatively reinforce bad actions (getting eaten by ghosts). The natural reward for a particular time step is simply the number of points earned. Nom Mon scales these values so that the amount of reward gained or lost is between 0.0 and 1.0 (so that it is in the range of the logistic function). In addition, Nom Mon adds a small bonus for every time step in which the player remains alive. Finally, when the player dies, Nom Mon penalizes the player. The reward at time step *t*, then, is determined by $r(t) = \frac{score\,(t)}{10{,}000} - 0.1 * (death) + 0.0001 * (alive) + 1.0 * (levelComplete)$ where score(t) is the score earned at time t, death is a Boolean variable which is True if the player died in the last time step, alive is a Boolean variable which is True if the player stayed alive in the last time step, and levelComplete is a Boolean variable which is True if the player completed a level in the last time step.

3.4 Experimental Design

The goal of the experiment is to test the efficacy of biologically-motivated learning techniques. A major issue with all of these techniques is the sheer amount of time they take to run.

The Blondie24 experiments, which relied only on a genetic algorithm, took over 6 months to complete[52] (although modern computers could likely complete this result several times faster). Tesauro's backgammon network with 20 hidden units took 200,000 games to gain respectable results. Playing 200,000 games of Nom Mon would take approximately 667 hours, or 28 days, to complete. In addition, the time cost of programming and debugging Nom Mon and the artificial intelligence techniques (containing more than 10,000 lines of code) was immense – over 9 months of programming. In order to allow greater experimentation with the network architecture and the parameters for temporal difference learning, the primary goal of the experiments was to test the efficacy of the biologically motivated methods when time was severely limited.

The first experiment tested the effect of input on the performance of the temporal difference network in Pac-Man. Tests were run twice each for thirty minutes of training time. The network used an architecture consisting of two hidden layers with 15 and 8 neurons. The chosen architecture is somewhat arbitrary, but reflects two general principles. First, two hidden layers generally produces a better approximation faster than a single hidden layer.[53] Second, even with the maximum number of inputs, there are the same number of neurons in the hidden layers as in the input and output layers. The results of these tests were used to decide which inputs would be used for the longer temporal difference learning test.

For the main experiment, all three learning techniques were tested for longer periods of time: temporal difference learning only, genetic algorithm only, and genetic algorithm with temporal difference learning. Due to time constraints, each test was run once for thirty-two hours, with a checkpoint at 8 hours to record the progress in the more limited time, and to allow a comparison between the shorter and the longer time periods. To address the limited time, strategies were developed for each learning method.

[52] (Fogel 2002) 256
[53] (Haykin 2009) 171

For experiments only testing temporal difference learning, a single feedforward neural network with the inputs from the best result in the previous experiment and two hidden layers was used. As before, the first hidden layer used 15 units; the second used 8. The network was trained until it exceeded the time limit by repeatedly playing Nom Mon with 3 lives, restarting every time it lost the game.

For experiments where only the genetic algorithm was tested, each neural network was tested once by playing a single game of Nom Mon with 3 lives. Unlike previous Pac-Man clones, Nom Mon is deterministic, so networks which do not learn will produce the same result during every run. A population size of 30 was used, allowing for approximately 240 generations of evolution over the entire 32-hour period. The initial population was created by generating a series of similar networks and converting them into genomes. Each network had two hidden layers with the number of neurons randomly chosen between 8 and 12. Between 4 and 8 features were randomly chosen for each network. Finally, the gamma, lambda, and learning rate parameters were set randomly in the ranges previously specified for each.

When testing temporal difference learning with the genetic algorithm, several factors must be balanced due to the sheer amount of time it takes to run many games of Nom Mon. If a greater proportion of time is spent on temporal difference training, a smaller population or fewer generations must be used. It was decided to only do temporal difference training on every 15th generation since only around four or five generations of TD training could be run. This also allows us to analyze how the two training methods can affect each other (that is, we can see if training a network with TD(λ) first allows the genetic algorithm to learn faster, or if TD(λ) is more effective after the networks have already been slightly optimized by the genetic algorithm). Other generations are evaluated as in the previous genetic algorithm experiment. Every 15th generation, however, the networks TD train for an certain amount of time and, at the end of that period, are evaluated on a single game with no learning (as in non-TD genetic algorithm experiments). Before breeding the selected parents, the new weights learned with temporal difference are copied back into the genome (the genome itself retains the same format, including any nodes that were not in the network due to disconnection; only nodes that were in the network have their weights updated). This allows the genetic algorithm to take advantage of the time spent in temporal difference learning. Even so, only 25 minutes could be afforded to each network for TD training, which allows the network to learn for around 120 games.

In both GA-based experiments, the starting population is a series of randomly generated neural networks, each with between 8 and 12 neurons in each of the two hidden layers. These networks are fully connected with random weights. The feature set was chosen to be between 5 and 10 random features from the list of possible features. The γ, λ, and α parameters were set randomly as described in 3.2 above. Both genetic algorithm experiments used a carry-over rate (keeping the best parent genomes from generation to generation) of 1/5th the total size of the population, rounded down.

The shorter time constraint allows us to see which technique is able to bootstrap from a random solution to a decent result most effectively. The longer time constraint allows us to see the effect of additional time on each technique. In addition, the experiment will answer the following questions:

1. What is the effect of adding additional time to the training for each technique? Does performance improve, or does it stay the same or worsen?
2. Is one technique significantly more effective in the limited amount of time? The longer amount of time?
3. Do any of the techniques scale better than the others with time? That is, is the best technique at 32 hours different than the one at 8 hours? Is the rate of improvement similar among all methods?

4. Results

4.1 Examination of Input Features

The first experiment tested the effect of various inputs on the performance of the temporal difference Pac-Man player during a short (30 minute) training period. Each test was run four times (twice using the hyperbolic tangent activation, and twice using the logistic activation) to ensure the tests were consistent; for clarity, only the best result from each is shown in the figures presented here (while the other tests are not significantly different, figures for all tests are included on the supplementary CD[54]). The network with "all" inputs used every feature. The numbers in parenthesis indicate how many network inputs the feature requires:

Dist. to nearest dot (1)	Dist. to nearest energizer (1)	Dist. to chasing ghosts (4)
Dist. to frightened ghosts (4)	Dist. to nearest junction (1)	Distance to fruit (1)
Ghost Speed (4)	Dot Ratio (1)	Player position (2)

First, a test was run to verify the effectiveness of removing two input features which were thought to be confusing or noisy: the dot ratio and the player position. While both have merits – the player position can tell the network whether the player is in a "safer" location (such as one of the T-shaped areas where ghosts have more trouble entering), and the dot ratio can give the network a hint about which direction leads to the greatest number of points – they both have significant issues. The player position is problematic because position is generally uncorrelated with score – once you have been to a particular location, returning to that location will not lead to additional points since the dots from that area have already been consumed. In addition, if the player eats a frightened ghost in that position, leading to a large number of points, the resulting points would be completely circumstantial, not a feature of the particular location. The dot ratio, on the other hand, displays a

[54] A listing of the CD contents can be found in Appendix A

very naïve view of the maze structure – for example, the dot ratio may indicate that 80% of the dots are found to the left; however, the fastest way to get *to* those dots may be by going up, down, or even right first. Thus, a network may often be given highly inconsistent rewards, even for the same value of the dot ratio.

Surprisingly, removing only one of the suspect features had almost no effect on the ability of the network to learn – these networks performed at the same level as the network which used all of the features (Figure 1). When removing both features, however, the network performs considerably better, attaining a maximum score more than four times greater than the networks using either of the suspect features. In addition, the network using neither dot ratio nor player position quickly learned to target energizers as they lead to the greatest possible score. The other networks appeared to wander aimlessly and died quickly.

Additional tests were performed removing other features which were thought to be somewhat inconsistent: ghost speed, ghost direction, and the distance to the nearest junction. The ghost speed is useful since it allows the network to tell if it can "outrun" a ghost in a particular segment; however, in early levels, Pac-Man will always be faster than the ghosts. The ghost direction should, in theory, be very useful since it doesn't matter nearly as much if Pac-Man gets closer to a ghost if that ghost is not moving towards the player. Due to the structure of the maze, however, the ghost may get closer to Pac-Man without actually turning in his direction (for example, if the ghost is moving parallel to Pac-Man). Distance to the nearest junction was thought to provide the best direction for the player to take when trying to escape; however, all junctions in the maze are very close – Pac-Man never has to travel more than five tiles to reach the nearest junction, so this input will constantly change as the player moves throughout the level.

All tests showed varying degrees of improvement during play; though the high scores were highly dependent on the starting scores, all showed positive trends between game 20 (when the random value (ε) reaches 0.01) and their final game (Figure 2). The simplest input vector – only the ghost distances to the nearest ghost, the distances to the nearest dot and energizer, and the distance to the fruit – showed the fastest improvement, from a score around 1,700 to over 2,500 after 108 games.

As a final test, a completely random player was run for 30 minutes. This random player (Figure 3) showed a wide variety of scores (as expected), with a mean score of 585.14, a median score of 575, and a standard deviation of 185.91.

4.2 Analysis of Three Biologically Motivated Learning Techniques

The main experiment aimed to find the best technique for teaching a neural network given a limited amount of time – just 32 hours, with a progress checkpoint after 8 hours. All three techniques exhibited varying degrees of learning, and all three showed significant improvement over a random player.

4.2.1 TD(λ) Experiment

Based on the results from the examination of input features above, the simple feature set using the nearest dot, nearest energizer, distances to both chasing and frightened ghosts and the distance to the nearest fruit was chosen for this experiment. The hidden and output layers used the hyperbolic tangent activation function, and parameters of $\alpha=0.05$, $\gamma=0.95$, and $\lambda=0.6$ were chosen. The ε parameter, which determines how often the program chooses a random move rather than the one suggested by the network, was set according to the schedule described in 3.4.3. At both 8 and 32 hours, the network was observed to play with a distinct though highly imperfect strategy, earning 2,800 points at its best (Figure 4).

After 8 hours and 3,390 games, the network achieved a ten-game running average of approximately 2,800 points. Between games 20 and 3222, the network consistently earned between 1600 and 2100 (aside from a few outliers); however, after game 3222, the network discovered a strategy which improved its score considerably. This strategy sent the player to capture all of the energizers first, often eating several ghosts on the way. The player showed little attraction to individual dots – once all the energizers were consumed, the player tended to wander randomly. In addition, the player does not appear to have developed any sense of ghost avoidance – often it causes its own death by running directly into a ghost, rather than attempting to avoid it.

After 32 hours and 8,998 games, the network achieved a ten-game running average of 1,820 points – a slight decline in performance after the additional training period. The scores trend also indicated that the network showed very little "innovation" during this later period – its score stayed within around 100 points for over 1,000 games. Though this result is expected due to the decreased value of ε as the number of games played increases, it appears that, rather than settling on a relatively high score and making minor improvements, the network fell into a pattern of relatively poor behavior and made no significant improvements. Because the probability of choosing a random action (and thus the probability of exploring the state space differently and finding new or interesting possible actions) was nearly zero for much of the later phase of training, the network was completely unable to try new solutions; in fact, the network was observed to have, with only minor variations, the same exact strategy for the last 4,000 games, or nearly half of the training time. This simple strategy, much like the one it exhibited after 8 hours, essentially led the computer player to eat all of the energizers first. Though it failed to actively chase the frightened ghosts, it did manage to eat some. When it reaches its final life, however, after all energizers have been consumed, the network fails at its remaining tasks – it is unable to avoid ghosts, and it appears to get confused often, running back and forth in a small area rather than collecting dots as quickly as possible. Still it is able to score 1,880 points, a score nearly 7 standard deviations higher than the mean score for the random player.

It appears that a major limitation of the TD(λ) experiment is the value of the ε parameter – as long as this parameter is relatively high (a value of at least 1 in 5,000, which lasted until around game 1,000 in this experiment, allows the network to choose a random action approximately every three games[55]), the network will try new strategies in every game and will therefore experience a wider variety of game states which, due to positive reinforcement, should lead to more accurate predictions and better overall strategies. However, when using the given schedule for ε, over half of the games played had little chance to explore the state space.

4.2.2 Genetic Algorithm Experiment

The genetic algorithm, with a population of 30 networks, completed 116 generations in 8 hours. The best network at that time achieved a significantly higher score (5,350) than that obtained by the neural network trained through TD(λ) (Figure 5). Unlike the TD(λ) algorithm, which may earn a very high score for several games but subsequently experience a drop in performance, the genetic algorithm used elitism, which ensures that the best members of the population are saved without modification.

The best genome used the feature set (ghost speed, ghost direction, dot, energizer, and fruit). Interestingly, the ghost distances – whether frightened or not – was not a part of the input set at all, though the dot and energizer distances were. The evolved network had a relatively simple architecture. It used the hyperbolic tangent as the hidden layer activation and linear as the output layer activation, and contained no feedback connections. Not all nodes are fully connected, and input 4 (the speed of the orange ghost) and input 9 (the distance to the nearest dot) have connections directly to the output neuron in addition to connections into the single 5-neuron hidden layer. In addition, only the first hidden neuron is fully connected to the input layer: the second neuron in the hidden layer is not connected to inputs 1, 2 (the speeds of the red and pink ghosts, respectively), or 5 (the direction of the red ghost), the third neuron is not connected to 6, 8, 9, or 10 (the directions of the pink and orange ghosts, the nearest dot, and the nearest energizer, respectively), the fourth neuron is not connected to 1, 2, or 3 (the speeds of all ghosts except the orange ghost), and the fifth neuron is not connected to 5.

[55] The player was able to make approximately 350 decisions in a single game before dying

It is difficult to speculate on the nature of the removed and added connections. It is logical to connect the dot-distance input directly to the output (the weight used was 0.0205), as collecting dots has a linear relationship to increased score; on the other hand, it seems unlikely that the connection from the orange ghost's speed to the output is of similar import. The purpose of the removed connections is similarly nebulous.

The best evolved network after 8 hours demonstrated a play style that was actually very similar to the best TD network. The main difference is that, once all energizers have been consumed, the network is able to collect the dots very efficiently. Due to this increased efficiency, this network is able to complete the first level, which gives it the opportunity to capture more energizers and ghosts, significantly increasing its score. Interestingly, however, the network clearly fails near the end of its last life when, though it is close enough to the energizer to turn the tides and eat several more dots, it simply stays in the same position (Figure 6).

After 32 hours and 273 generations, the genetic algorithm had produced a slightly improved network able to score 5,770 points (Figure 5). This evolved network plays essentially the same game as the previous network during the first level, but rather than staying still in the upper-right corner near the end of its last life, the network does capture the energizer and continue collecting dots. It's death is remarkably similar, however – rather than running for the energizer at the top left of the screen as the ghosts close in, it stays in the corner until the ghosts catch up to it (Figure 7).

The architecture of the network indicates that evolution only made very small tweaks over the last three quarters of the training period. The network used the same features and activation functions, but with a slightly different network architecture. Oddly, the output neuron was connected to the input bias. The second input was disconnected from inputs 1, 2, and 5; the third from 2, 8, 9, and 10; the fourth from 1 and 2; and the fifth was fully connected. No additional connections, other than the input bias, were skipped in the hidden layer.

Despite the clear weakness of the feature set, the top networks from both the 8 hour and 32 hour sets use the same set of features, which is clearly not ideal since it lacks any information about where the ghosts are. It is for this reason that the evolved networks never learned to avoid ghosts – the early levels of Pac-Man keep the ghosts frightened long enough after eating an energizer that it is still possible to complete a level. When started at later levels, when the blue time is non-existent, the best network was observed to fail much more quickly, eating just 98 dots and 2 energizers before losing all three lives.

4.2.3 Genetic Algorithm with TD(λ) Experiment

The final experiment tested a combination of the previous two experiments. In the first generation of the genetic algorithm, and every 15th generation thereafter, each evolved network was tested by training it for 25 minutes using TD(λ) and then the resulting network's fitness was evaluated as in the previous genetic algorithm-based experiment. After training, the changed weights were copied back into the genome. Other generations were evolved without the TD(λ) step. It was hoped that the combined approach would improve on the results obtained from the separate tests – since the TD(λ) algorithm is very sensitive to starting states (as shown in the 30 minute experiments, some networks achieved a score of 1600 by game 20 and continued to improve), it was hoped that the genetic algorithm could start to optimize the network to "bootstrap" the networks training using TD(λ).

The overall results were heavily mixed. After 8 hours and 16 generations (two TD(λ)-evaluated generations), the top networks improved from a maximum score of 1,270 to 1,560. However, the result after generation 15 (a top score of 3,310) indicates that networks which play strongly in normal conditions were unable to continue to learn using TD(λ). The best evolved network at generation 16 was a fully-connected feedforward network with just two layers with the logistic activation function for the output layer using the feature set (ghost position, ghost speed, ghost direction, dot, energizer, and junction). It showed perhaps the strangest style of play – during its first two lives, the network collects just 24 dots and no energizers, essentially staying in the same part of the maze. In its third life, however, the network collects all four energizers and 112 dots, but no frightened ghosts (unsurprising since it has no inkling of their location from its set of input features).

After 32 hours and 61 generations (5 TD(λ)-evaluated generations), the best evolved network had a maximum score of 2,550, higher than the network trained only using TD(λ) for 32 hours (Figure 8). This network had three layers using logistic activations for both the hidden and output layer and the same input features as the previous network. It appears that this version of the genetic algorithm, as in the previous experiment, was unable to evolve the feature set. This network's output layer exhibited several interesting connections: in addition to connections to the hidden layer, the inputs for the nearest dot, energizer, junction, distances to the blue and orange ghosts, the direction of all of the ghosts, and the speed of the orange ghost were connected directly to the output layer. The hidden layer was sparsely connected with the inputs: the first hidden neuron was connected to all but the 6th and 11th inputs (the pink ghost's direction and the blue ghost's speed) and the bias; the second hidden neuron was disconnected from distances to all ghost distances and the red ghost's speed (inputs 5, 6, 7, 8, and 9); the third and fourth hidden neurons were both only connected to the red ghost position and the nearest energizer (inputs 1 and 14); and the fifth hidden neuron was connected only to the two inputs above and the directions of the blue and orange ghosts (inputs 1, 7, 8, and 14). The connection pattern of the third and fourth hidden neurons is especially interesting – it appears that the network architecture innately attempts to balance the drive to eat the energizers and the need to avoid the most persistent of the ghosts.

The actual behavior of the evolved network was significantly improved over its eight-hour predecessor. Like the TD-only network, this evolved network immediately aims for the energizers; however, unlike the TD-only network, this evolved network is able to eat dots fairly effectively after the last energizer is gone. Like all other networks, it fails to effectively evade the ghosts as it collects dots, leading to frequent deaths.

This method shares the weakness inherent in the genetic algorithm-only experiment: the feature vector is unable to evolve with the network weights and architecture. In addition, the network trains much more slowly due to the time spent on TD(λ) training (each TD(λ) training generation took around 6 hours). While the networks' scores often improved quickly during generations where the networks were only evaluated without learning, TD(λ) generations uniformly resulted in reduced scores. Interestingly, the magnitude of the score drop increased during the first few generations, but then dropped quickly – after the second TD(λ) generation, the score dropped by around 51% (from 3,170 to 1,560), after the third it dropped by around 69% (from 3,410 to 1,060), after the fourth it dropped by around 53% (from 5,240 to 2,440), and after the fifth the score dropped by just 9% (from 2,790 to 2,550). It was thought that the learning parameters – the discount rate (γ), the trace-decay (λ), and the learning rate (α)— would converge to values well-suited to TD(λ) learning, but these values were identical (0.83, 0.71, and 0.15, respectively) after both the 8 hour and 32 hour experiments. It appears to be the case that without continuous selective pressure on those attributes (generations run without TD(λ) did not require any learning parameters), the genetic algorithm failed to adapt them.

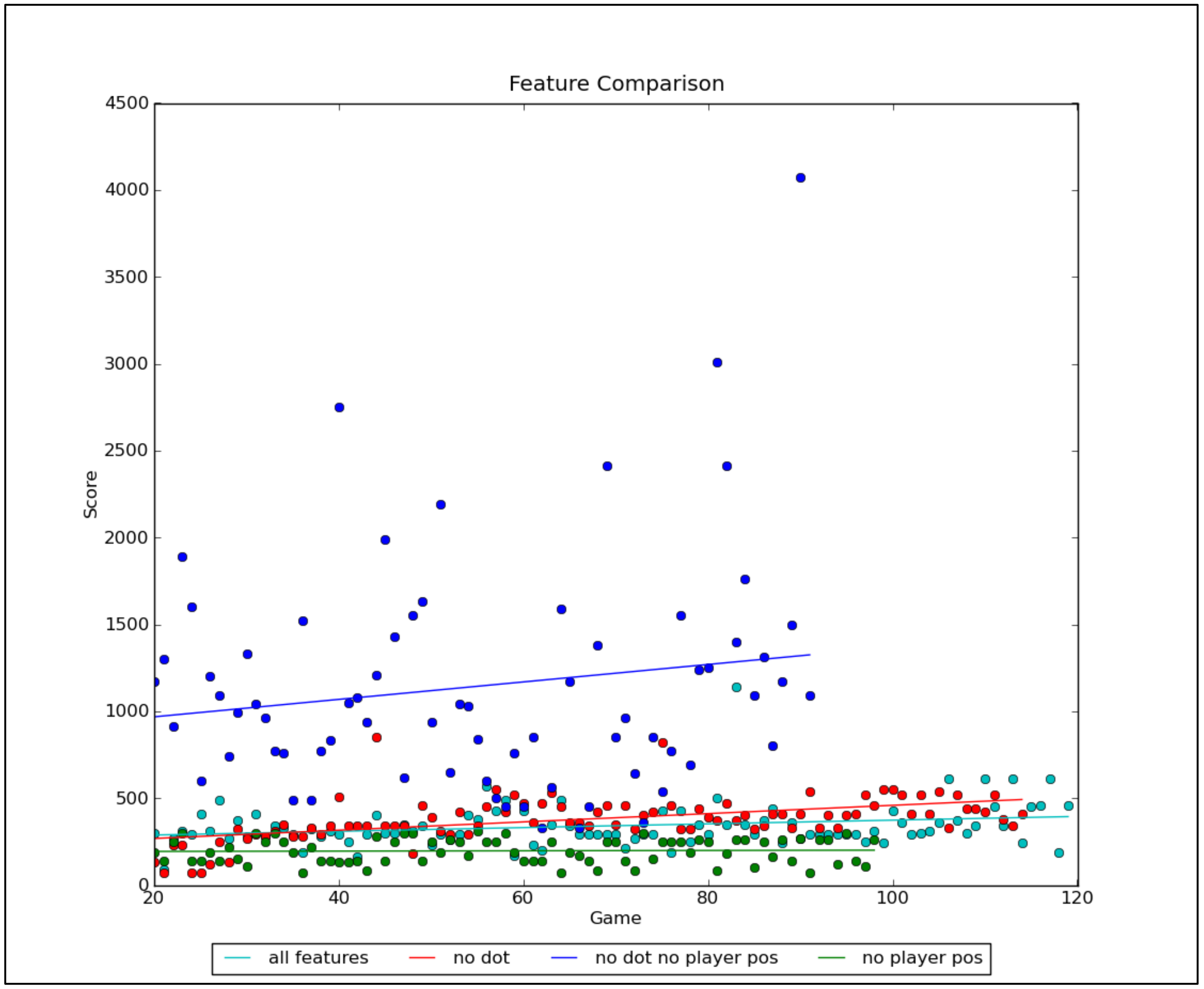

Figure 1: Score comparison between networks using all features and those removing the dot ratio and the player position

The network with neither the player position nor the dot ratio performed much better than networks that used either of these features.

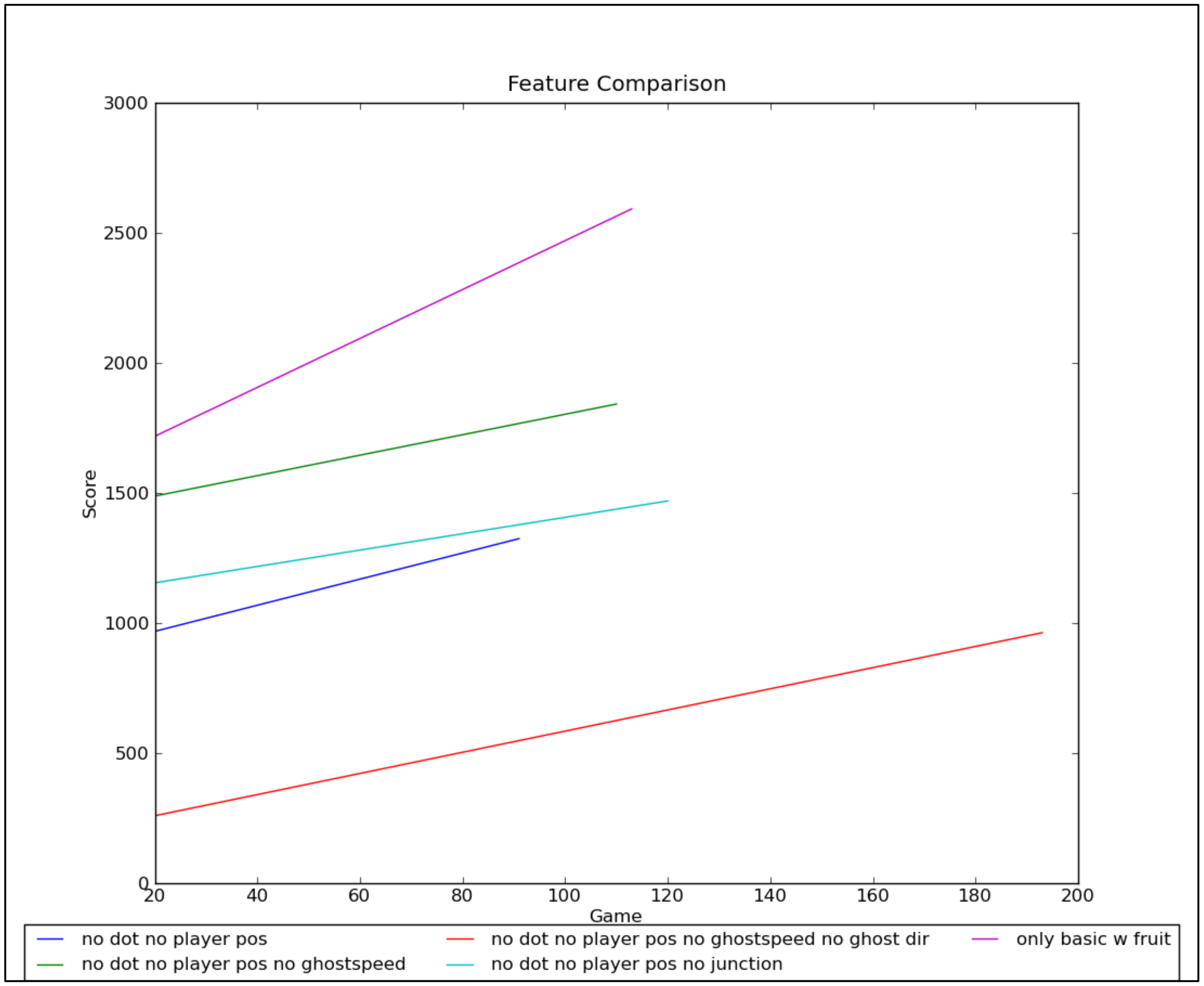

Figure 2: Comparison between networks using various features

All feature-sets allowed the network to improve, though networks with higher starting scores did better overall.

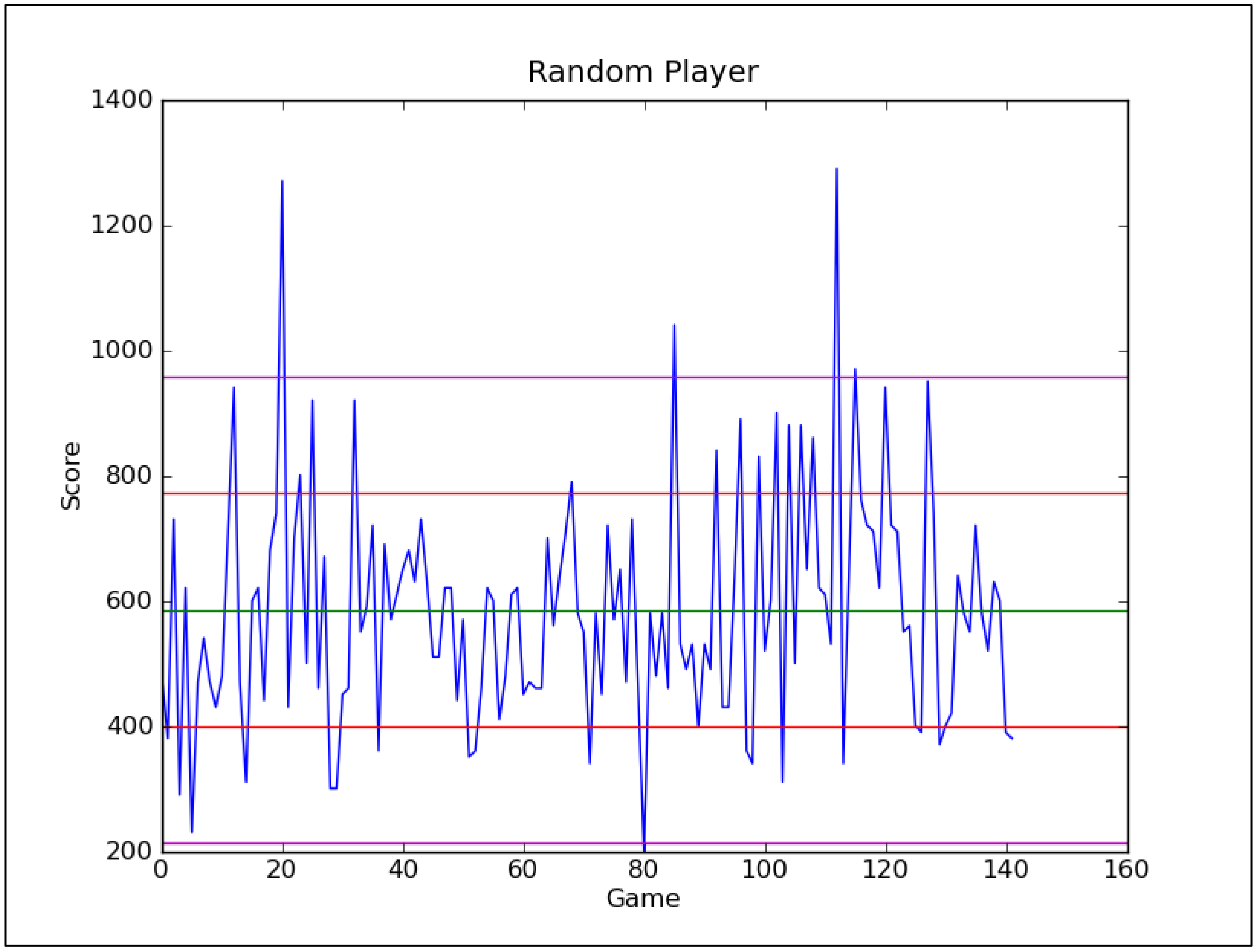

Figure 3: Scores for the random Pac-Man player

The random Pac-Man player achieved very low scores. The green line indicates the mean, while the red and purple lines indicate one and two standard deviations from the mean, respectively. All 8- and 32-hour tests created neural networks that learned to play significantly better than this random player.

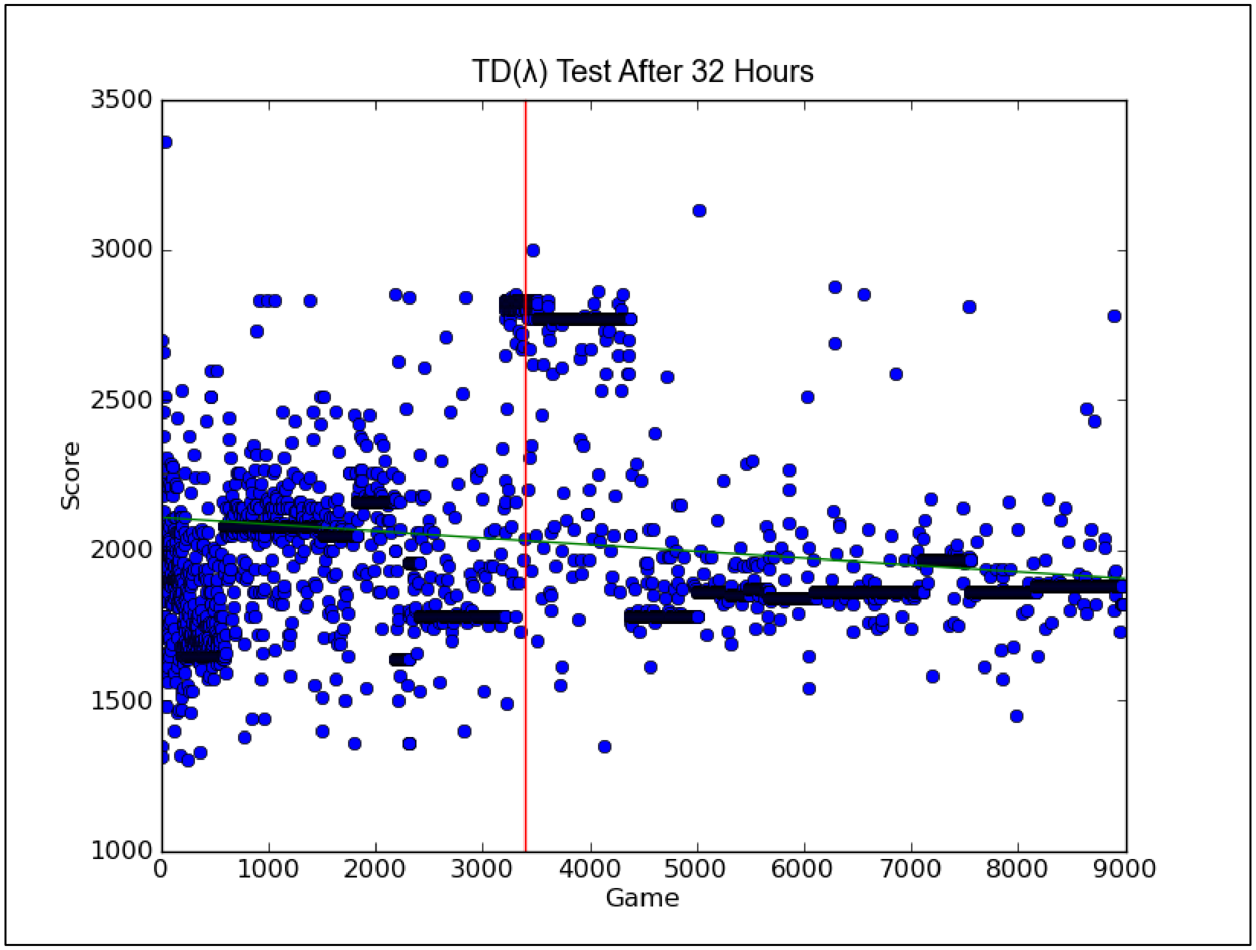

Figure 4: Scores for the TD(λ) test

Each dot indicates the score of a particular game. The dark lines that form are simply a large number of dots clustered at approximately the same score, indicating periods of relative stability. The green line indicates the linear regression line, and the red line indicates the end of the eight-hour checkpoint.

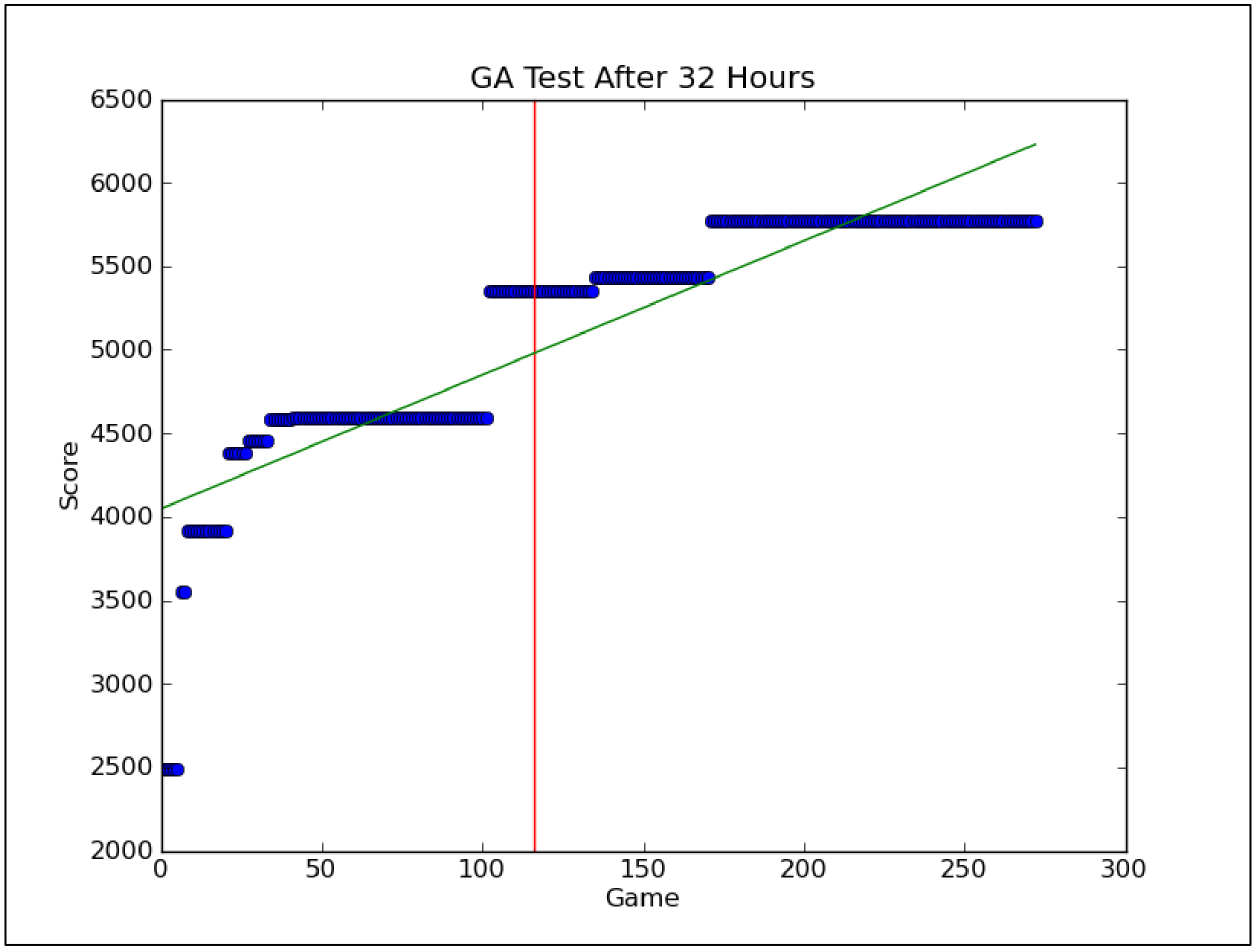

Figure 5: Scores for the genetic algorithm test

The genetic algorithm achieved the highest score of all tests and completed the first level. Again, the blue dots represent the top score during that generation, the green line represents the linear regression line, and the red line indicates the end of the eight-hour checkpoint.

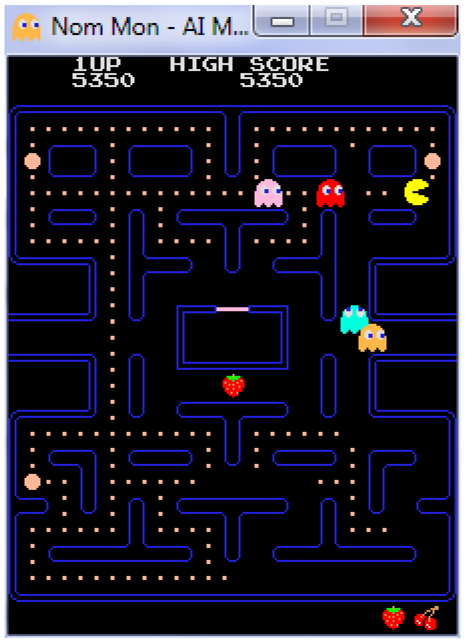

Figure 6: Pac-Man's death at the end of the 8-hour GA experiment

Pac-Man oscillated at this position rather than eating the energizer, leading to his premature death.

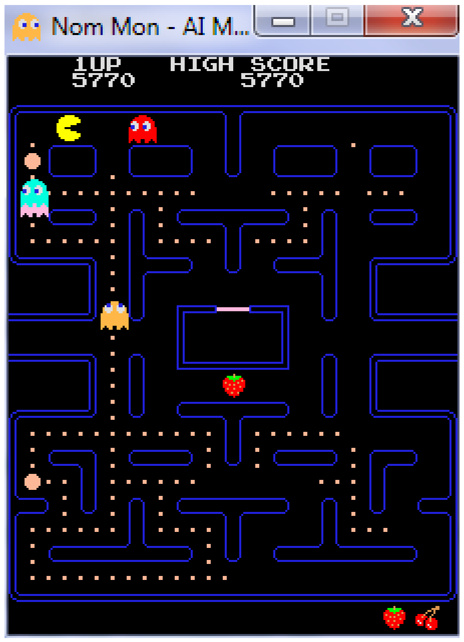

Figure 7: Pac-Man's death at the end of the 32-hour GA experiment

Pac-Man once again has the chance to capture the energizer and prolong his life, but oscillates in the same position until death.

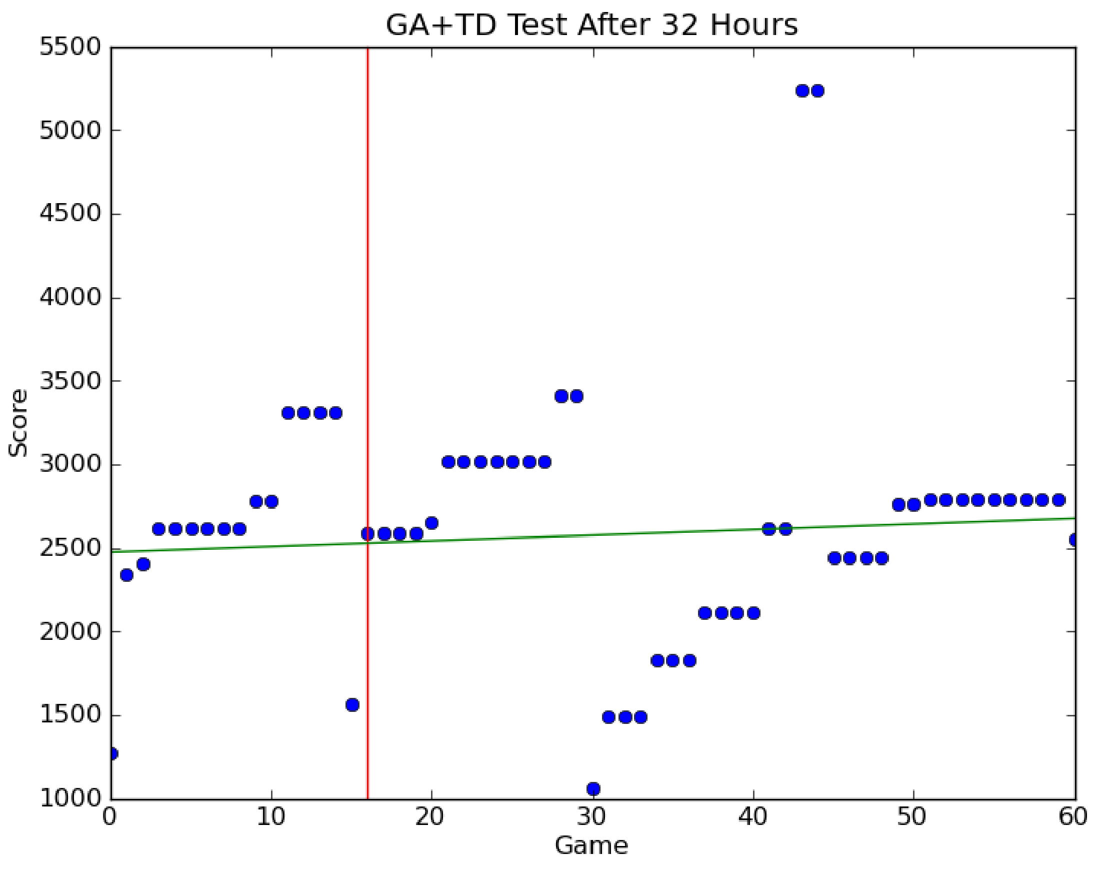

Figure 8: Scores for the genetic algorithm with TD(λ)

The experiment indicated a more slowly-increasing score, as indicated by the green regression line. The scores decreased significantly every 15th generation when the networks were tested using TD(λ) instead of the standard one-game test with no learning.

5. Conclusions

The data produced as a result of the experiments in this thesis indicate that all three biologically-motivated methods for teaching a neural network – TD(λ), genetic algorithm, and genetic algorithm with TD(λ) – are capable of learning, at least to some extent, in a complex video game environment which also presents a challenge to human players. The genetic algorithm was even able to develop a network which is able to consume all 244 dots in the first level, a result comparable to previous research using simpler Pac-Man environments.

The results were, in some ways, disappointing – no networks developed even the simplest ghost-avoidance strategies, and none made use of special maze features. In some ways, these observations stem directly from the choice of input features for the Pac-Man program. Using maze features properly requires that the player knows its own position as well as the positions of the chasing ghosts. The input features used in this thesis, however, only gave the distances to the ghosts; these distances, however, completely ignore the maze structure (for example, the tunnel and the T-shaped junctions above and below the ghost pen can allow a clever player to escape chasing ghosts, but only if the ghosts are in a certain location relative to the player).

5.1 Future Work

While the experiment clearly showed that including certain input features resulted in significantly worsened learning, it is unclear exactly what the ideal input features are for a neural network that can play Pac-Man . The features tested in this thesis provide a very basic starting point, but any future work should attempt to design features which incorporate information about the maze's structure. In addition, while a previous result showed some promise with raw inputs, it would be interesting to extend this idea to a more complete representation of the board. For example, the "window" metaphor for the input from Gallagher et al., could be extended to all important parts of the maze (for example, the area near the energizers could be shown), or sections of the maze could be summarized (for example, the number of dots, energizers, and ghosts in an area could be given).

In addition, the ε (randomness) parameter could be modified during play to allow the network to continue to explore the state space as it improves its play. A previous paper, which applied SARSA(λ)[56] (a variation on TD(λ) which approximates the value of a (state, action) pair rather than just the value of the state) to Pac-Man, evolved the ε parameter after each time-step using an adaptive genetic algorithm.[57] The programs which adapted the exploration rate during the game based on the player's performance showed scores approximately 41% higher than the player with a constant random exploration rate, and were able to achieve approximately half of the total possible score on the level designed by the authors.

It appears that the major weakness in the genetic algorithm experiments is the simultaneous evolution of both the weights and the features. Recall that the genome representation defines a dendrite using node number. The number of a node is determined simply by consecutively numbering each node as it is encountered when processing the genome. However, if a feature is added or removed, the number of inputs will change, and therefore the numbering of all nodes occurring later in the genome will change. Thus, what once may have been a connection from the first hidden node to the output node could now be a connection from an input node to the output node. The weight, however, was evolved based on the connection from the hidden node. Thus, the weight is likely to be invalid, and the network performance will drop. A future experiment should more closely correlate the feature enumeration with the weights; for example, when a new feature is added, the existing connections with the same or higher node numbers could have their numbers automatically incremented. Then the new feature could have connections created to random nodes with random weights.

[56] The authors used a table mapping (state, action) pairs to a value, rather than a neural network, as we do in this thesis

[57] (Galway, Charles and Black 2009) 38

It is notable that, while feedback connections were allowed in the design and evolution of the genome, no high-scoring network used them. This unfortunately indicates nothing about the possibility that such connections can improve the capabilities of a network which interacts in an environment which changes through time. Future tests should analyze this network construct more rigorously in the context of an environment that changes through time.

A different experimental setup for the combined experiment may have yielded better results. Rather than starting with a TD(λ) generation, the genetic algorithm could have begun optimizing weights for several generations. After initial training, the genetic algorithm would simply use TD(λ) for all additional generations. This would be more time-consuming, but with enough generations it would force the genetic algorithm to actually optimize the network structure and features to the problem of learning with TD(λ), as well as optimize the weights themselves.

Appendix A

This appendix contains a listing of the files included on the accompanying CD. This CD contains the Python source code used to create the results found in this thesis, as well as automatically generated graphs and binary files which contain the data necessary to inspect the results of the thesis (the binary files contain Python objects which were serialized to disk during execution after 8 and 32 hours for each experiment).

The Python concept of packages, which these programs use, is directly related to the hierarchy of folders. When moving the code, be certain that all .py files are in the same relative position to each other – the aiExperiment.py file in the root directory and two source directories, ArtificialIntelligence and Nom_Mon.

Note that some of the code for this project assumes a writable file system relative to the current path (to save the objects during run time to help avoid data loss). It is therefore recommended that all program files be copied to a drive location that you have permission to write to. Many programs also erroneously assume that existing files should be overwritten. It is therefore additionally recommended that data files (.ann, .gga, .png, .sav) be backed up before re-running the experiments.

File or Directory Name	**Description**
/aiExperiment.py	Supporting code to run all experiments performed in this thesis
/chromosomeTest.py	Unit test for EvolveNomMonANN.py
/pydoc.py	Python documentation program
/pydocgui.pyw	Python documentation graphical interface
/testRunner.py	Simple program to run a single result at human speed

/ArtificialIntelligence/	Package of artificial intelligence modules
ActivationFunctions.py	Activation functions for the neural network
ANNDriver.py	Sample code and tests for the artificial neural network
ANNfile.py	Code to save neural networks to disk
annGUI.glade	XML file containing GUI information for annView
annView.py	Program to display and manipulate ANNs graphically
ArtificialNeuralNetwork.py	Implementation of a flexible, object-oriented ANN
EvolveBooleanANN.py	Supporting genetic algorithm functions for a simple experiment evolving a neural network
EvolveNomMonANN.py	Supporting genetic algorithm functions for evolving neural networks to play Nom Mon
EvolveTSP.py	Supporting genetic algorithm functions for evolving solutions to the travelling salesman problem
GADriver.py	Sample code and tests for the genetic algorithm
GeneticAlgorithm.py	A generic, object-oriented Genetic Algorithm
SelectionFunctions.py	Selection functions (roulette and random) for the GA
TicTacToe.py	A simple test for the TD(λ) algorithm that teaches a neural network to play Tic-Tac-Toe. Used to verify the implementation of TD(λ).
/ArtificialIntelligence/images/*	Images used by the annView graphical program
/Data/*	Data files used by Nom Mon to generate the maze and the pseudo random number generator
/Graphics/*	Graphics files used by Nom Mon
/NomMon	Package for the Pac-Man replica, Nom Mon
mapCreator.glade	XML file containing GUI information for mazeCreator

mazeCreator.py	Program to create mazes for Nom Mon
Nom_Mon.py	A replica of Pac-Man that allows for AI input and learning with a TD(λ)-based neural network.
/THESIS/	Files created for the thesis specifically, including results, charts, and graphs
exp1.png	Graph comparing scores between all features and programs run after the removal of the dot ratio and player position
exp2.png	Comparison of other features
GA_8hr_death.png	Screenshot showing the AI's death after 8 hours in the GA-only experiment
GA_32hr_death.png	As above, but after 32 hours
ga_exp.png	Graph showing the scores over time for the GA-only experiment
ga+td_exp.png	As above but for the GA+TD experiment
random.png	Graph showing the performance and statistics of the random player
td_exp.png	Graph showing the scores over time for the TD-only experiment
/TEHSIS/Results/	Folder containing raw results
Features Test	Folder containing automatically generated results for the tests for each feature
*.sav	TestResult objects saved to disk for later inspection (see B.4)
*-scores.png	The raw scores for each experiment in a graph, not including the first 20 games where $\varepsilon>0.01$
*-averages.png	Graphs showing 10-game running averages

*-final.txt	Text file containing all scores and other statistics
GA Test	Folder containing automatically generated results for the GA test. See above, under Features Test, for descriptions of each type of file.
-checkpoint	Files with checkpoint in the title were snapshots taken after 8 hours. The others will have final in the title.
GA+TD Test	See description above, but for the GA+TD test.
Random Test	See description above, but for the random player. No .sav files are kept since no learning occurred.
TD Test	See description above, but for the TD test.

Appendix B

This appendix contains a brief guide to using the source code included with this program. The guide assumes basic familiarity with the Python programming language. This includes a brief introduction to the ArtificialNeuralNetwork.py module, the GeneticAlgorithm.py module, the Nom_Mon.py module, and the aiExperiment.py module. The first two are object-oriented libraries that provide support for their respective AI techniques. Nom_Mon.py implements the replica of Pac-Man, Nom Mon, and the AI object which supports interfacing between ArtificialNeuralNetwork.py and the Nom Mon game. aiExperiment.py is the code used to generate the results found in this thesis.

All Python code is compatible with Python 2.6 and greater versions of Python 2. Python 2.5 and below and Python 3.0 and above cannot run these programs. The artificial intelligence themselves depend only on Python's standard library. Example code and tests all require matplotlib. The graphical ANNView.py and mazeCreator.py both require GTK+ and Glade installed. Nom Mon requires that Pygame be installed. It is recommended that users install Pscyo as well, though this is not required. The programs were tested on Windows 7 (64 bit, but using 32 bit Python 2.6.5) and Ubuntu 10.04 (64 bit using 64-bit Python 2.6.5). Other platforms using Python 2.6, including Mac OS X and other Linux/Unix, should work properly. Note that Psyco does not work on 64-bit platforms.

For more detailed, technical information about each module, run **pydocgui.pyw** in the root directory of the CD (you may need to allow it through your firewall) and press "Open in Browser." Look for the package names (Nom_Mon and ArtificialIntelligence) and click on any file or package there to view its documentation. This program is included by default with Python, and thus ships with its license. All other programs are distributed under the MIT license, which is included in each source file.

If you find this documentation to be incomplete in any way, I am available to answer any questions about the code specifically by email at daniel.goldsmith@ncf.edu. Please include a short, descriptive Subject: field, and describe your question clearly and concisely.

B.1 GeneticAlgorithm.py

Description

This module allows one to solve arbitrary problems using a genetic algorithm. For all problems, a function which translates a genome into the object to be tested (which is accepted by the fitness function) and a function which assesses the fitness of a given (object, genome) pair. Three types of chromosome are implemented: binary, real, and combinatoric. Binary operates on strings of bits as chromosomes. Real operates on lists of floating point numbers. Combinatoric operates on ordered lists (this is useful for, e.g., the travelling salesman problem). Only binary chromosomes have been thoroughly tested, though simple test cases are included for all three chromosomes. Note that the binary chromosome class contains several modifications specific to this particular experiment; in particular, it treats chromosomes with a block size of 64 (the first chromosome in this experiment) differently from those with a block size of 32 (the second chromosome).

The genetic algorithm is not well optimized for very large population sizes; it is best used in problems where the fitness evaluation takes the majority of the time.

Interface

```
class GeneticAlgorithm(object):
    """
    This class implements a generic Genetic Algorithm.
    When creating the algorithm, you must specify what the genome will
look like, mutation rate (or 0 for no mutation), selection function,
and fitness function. When running the algorithm, a fitness target or
target number of generations can be supplied as a stopping point.
    """

    def __init__(self, genomeSpec, mutationRate=0.05,
                  crossoverRate=0.95,
                  populationSize=1000, carryOver=0,
                  selectionFunction="roulette", minimize=False,
                  fitnessFunction=None, genomeFunction=None,
                  saveDirectory=None)
     def run(fitnessTarget, minGenerations=10, maxGenerations=2000)
```

1. genomeSpec: A list of tuples describing the format of the genome's chromosomes (if the genome will have one chromosome, this should be a list with one tuple). The first item in each tuple is the chromosome type – either "binary," "real," or "combinatoric." The second item is the type of crossover – either a number to use that number of crossover points, or "uniform" to take genes alternately from each parent. The next items are dependent on the type of chromosome.

    a. For binary:

        i. Block size: the number of bits in each block. Blocks are operated on as a unit by crossover and mutation

    b. For real:

        i. Size: the number of real numbers in each chromosome

        ii. Range: the initial lower and upper bounds for the genes

        iii. Mutation: the minimum and maximum magnitude of mutation. When selected for mutation, a gene will be changed by a random magnitude in this range. The sign of the change is random.

    c. For combinatoric:

        i. Size: the number of genes in each chromosome

        ii. Alphabet: a list of strings that uniquely identify each item in the set you are trying to optimize (for example, when creating cities in the travelling salesman problem, you can just provide a range of numbers as strings, where each number identifies a city).

2. Mutation Rate: the probability that a gene will be mutated.

3. Crossover Rate: the probability that two parents will be crossed over; if crossover is not performed one of the parents is returned unchanged. If crossover is performed but fails, one of the parents is randomly selected, mutated with double the normal mutation rate, and returned.

4. populationSize: the number of individual genomes to be evolved

5. carryOver: the best genomes from the previous generation can be kept without modification to prevent the loss of a good solution. Genomes that have already been evaluated will not be evaluated again. This number tells how many of the top genomes will be kept.

6. selectionFunction: either "roulette" (for fitness-proportionate selection) or "random" to select the genomes to breed randomly.

7. Minimize: set to True if you are trying to minimize, rather than maximize, a value (as in the travelling salesman problem where the goal is to minimize the total path).

8. fitnessFunction: a function that takes as arguments the object a genome represents and the genome itself. Must return an integer or floating point value describing the fitness of the given object. For custom problems, you must write this function yourself.

9. genomeFunction: a function that takes as an argument a genome and returns the object it represents (suitable to pass to fitnessFunction). For custom problems, you must write this function yourself.

10. saveDirectory: a directory where saved instances of the genetic algorithm will be kept (if None, the genetic algorithm will not automatically save itself). This is useful for preventing

data loss since genetic algorithms are slow. The directory should be empty. Genomes are saved every generation, but only every 10th generation is kept. Automatic saving will only occur if the genetic algorithm is run via the GeneticAlgorithm.run() function.

Usage

Generally, it is sufficient to create the genetic algorithm and simply run it. The run() function takes two arguments: the fitness target is the fitness at which evolution will be considered satisfactory. The function will terminate evolution after maxGenerations have been run. You can also provide minGenerations – even if the fitness target is met, the function will run at least minGenerations generations.

For more information and sample code, the file **GADriver.py** has several example genetic algorithm tests. Example implementation of fitness functions and genome functions are given in EvolveTSP.py, EvolveBooleanANN.py, and EvolveNomMonANN.py.

B2. ArtificialNeuralNetwork.py

Description

This module allows one to create neural network with arbitrary network architectures. Both backpropagation learning and temporal difference learning are available.

Interface

```
class ArtificialNeuralNetwork(object):
    """
    A class that implements a Multi-Layered Perceptron
    The ANN contains "layers" which, in turn, contain
    "neurons" which, in their turn, contain connections
    to other neurons.
    """

    def __init__(self, layerSpecifications, standardMLP=True,
weightMatrix=None)
```

```
    def runOnce(self, inputList)

    def temporalDifferenceTrain(self, currentReward, currentPrediction,
lastPrediction, learningRate=0.1, lambdaValue=0.2, gammaValue=0.9)

    def backpropagationTrain(self, inputList, expectedOutput,
                             learningRate1=0.12, learningRate2=0.02,
                             momentum=0.0, targetError=0.01,
                             maxCycles=10000, minCycles=100)
    def save(self, filename, overwrite=False)
```

Constructor:

1. layerSpecifications: a list of lists that describe each layer. Each internal list tells the number of neurons in that layer and the activation type used for the layer. Note that the input layer (the first layer) always uses linear activation, regardless of specification. In addition, the input layer will automatically have a bias neuron created. Example specification for a three-layer network with 3 inputs, 5 hidden units and 1 ouptut, with the last two layers using the hyperbolic tangent activation:

```
layerSpecs = ((3, “linear”), (5, “tanh”), (1, “tanh”))
```

2. standardMLP: if True, each layer will be fully connected with the next layer. In this case, weightMatrix should not be provided (it should be None).

3. weightMatrix: if standardMLP is False, this is a list of lists of matrices. Each list represents a layer, and each list internal to that represents a neuron. Each matrix is then a neuron’s weight matrix. Each list in that weight matrix represents a layer, and each item in that list represents a connection to that neuron (a value of 0.0 means that there is no connection to that neuron). Neurons in the same layer must not be interconnected.

runOnce():

1. inputList: the list of inputs to provide to the network. You must provide exactly one input per input neuron. The result is returned as a list, one element per output neuron

termporalDifferenceTrain():

1. currentReward: the reward earned in the last time step
2. currentPrediction: the predicted reward in the last time step
3. lastPrediction: the predicted reward in the time step previous to the last one
4. learningRate: the percent of the error that weights should move; lower values help prevent oscillation.
5. lambdaValue: the λ eligibility trace parameter.
6. gammaValue: the γ trace decay parameter.

backpropagationTrain():

1. inputList: a list of lists of inputs to the network. Each list is a set of inputs to be trained on. Each internal list must provide exactly one input for each input neuron.
2. expectedOutput: a list of lists of outputs for the network. Each list corresponds to the input list at the same index from inputList.
3. learningRate1: the learning rate to be used at the beginning of training
4. learningRate2: the learning rate to be used later in training. The algorithm automatically switches to this learning rate when log(error, targetError) > 0.8. To use a constant learning rate, set learningRate2 equal to learningRate1.
5. momentum: the percentage that a previous weight change should affect the current weight change (i.e., 0.8 means that 80% of the previous weight change will be added to the current weight change). This is generally 0 as it was found that anything other than absurdly small values like 0.0001 would cause the algorithm to diverge.

6. targetError: When the algorithm detects that the mean squared error across the entire training set is less than or equal to this error, the learning terminates.

7. maxCycles: the maximum number of training cycles that will be performed. If the target error is not reached by the time this many training cycles has been completed, learning will terminate.

8. minCycles: the minimum number of training cycles that will be performed. Even if the target error is reached the algorithm will continue to run for at least this many cycles.

save():

1. filename: the file name under which the network should be saved. Include either a full path (e.g., "/home/me/Desktop/test.ann") or a relative path ("test.ann").

2. overwrite: set to True if you want to automatically overwrite the given filename. Set to False to raise an exception if the given filename exists. This is not interactive (it won't ask for a new filename if the given filename is incorrect); you can write your own interactive function by catching the raised ANNfile.ANNFileError exception when overwrite is set to False.

Usage

For backpropagation, it is sufficient to simply create the network and run backpropagationTrain() as described. You can test the network with runOnce(). Temporal difference training requires that you write supporting code for the rest of the agent (e.g., interacting with the environment, calculating the reward, testing different actions, etc). A graphical interface is provided in **ANNView.py**. This is a very simple program that allows one to inspect the network weights and architecture and test and train simple patterns. You can also get basic information about the neural network simply by printing it; for example:

```
>>> print ann
```

```
Artificial Neural Network consisting of 4 layers and 70 neurons.
**Layer 1 with 11 neurons & 1 bias using activation linear.
**Layer 2 with 15 neurons using activation tanh.
**Layer 3 with 8 neurons using activation tanh.
**Layer 4 with 1 neuron using activation tanh.
This network has been trained to a MSE of 0.01315823
This network has 308 connections.
```

For more information and sample code, the file **ANNDriver.py** has several example artificial neural network tests.

B3. NomMon.py

Description

NomMon.py implements a close replica of the arcade game Pac-Man. The game supports human testing (by simply running the script) or AI play through the AI class. The AI class will be described here.

Interface

```
class AI(object):
    def __init__(self, ann, epsilon=0.01, lookahead=2,
                 learningRate=0.08, lambdaValue=0.3, gammaValue=0.9,
                 features=('dot','energizer','ghost_pos',
                          'fright_pos','ghost_speed','ghost_direction',
                          'junction','player_pos','fruit','dot_ratio'))
```

1. ann: this should be an ArtificialNeuralNetwork object, as explained above.

2. epsilon: this is the probability of choosing a random action rather than the perceived "best" action

3. lookahead: the input provided to the network is the player moved in the tested direction and the distances to the ghosts and dots are computed. Lookahead is the number of tiles the player should be moved in each direction.

4. learningRate: the learning rate to be used by the TD algorithm

5. lambdaValue: the λ eligibility trace parameter

6. gammaValue: the γ trace-decay parameter

7. features: a tuple of strings indicating the features to use. The features can be given in any order (they are always presented in the order shown to the network).

Usage

Generally, to use Nom Mon in AI mode you should first create a neural network as instructed in the previous section. Then create an AI object using the created neural network as the first parameter and a list of features as a keyword argument. To use a schedule for epsilon (that is, a value for epsilon that changes after each game), you must manually update the epsilon value of the AI. You can then manually run a game:

```
ann = ArtificialNeuralNetwork(((23, "linear"), (20, "tanh" ),
                               (1, "tanh")), standardMLP=True)
ai = NomMon.AI(ai)
# set learning to False to watch the AI play
game = NomMon.Game(ai, learning=True)
# starts a single game and returns the score
# set game.clockTick = 60.606060606 to run the game at normal speed
# (the game normally doesn't limit the game speed so that as many games
# as possible can be played)
score = game.attractScreen()
```

NomMon.py also provides a helper function, playAIGame, which allows you to specify the AI object, number of games to play, and whether the AI should learn while playing:

```
playAIGame(ai, runs, learning=True)
```

ANNDriver.py provides an example of learning with NomMon using several networks.

B4. aiExperiment.py

Description

This is the file from which all experiments from this thesis were run. To duplicate the experiments, simply run

```
python aiExperiment.py
```

Before running this file, make sure that you are on a writable file system location, as described in Appendix A. Note that the function continueTests() has not been implemented due to time constraints.

This file also contains the definition for the TestResult class. All .sav files on the CD are TestResult objects, which include a copy of the artificial intelligence (whether the neural network or the genetic algorithm and its population). There is enough information stored in the .sav files to restart tests by running them for the remaining amount of time. See the documentation for TestResult for more details. This should only be necessary should the computer or software crash during a run. Note that a fairly large amount of memory may be required, especially during the GA-only experiment. It is recommended that you have at least 300 MB of available, free RAM before running this program. In addition, each test will use all available computing resources in a single process. Thus, it is highly recommended to run the program on a computer with at least two processor cores.

Bibliography

Burroughs, Joshua. "Impact of Redundant Data on Evolution of Neural Networks." *B.A. Thesis. New College of Florida*, 2005.

Burrow, Peter, and Simon Lucas. "Evolution versus Temporal DIfference Learning to Play Ms. Pac-Man." *IEEE Symposium on Computational Intelligence and Games*, 2009: 53-60.

Dayan, Peter, P. Read Montague, and Wolfram Schultz. "A Neural Substrate of Prediction and Reward." *Science Magazine*, 2007: 1593-1599.

de Castro, Leandro. *Fundamentals of Natural Computing.* Boca Raton: Chapman & Hall, 2006.

Elman, Jeff. "Finding Structure in Time." *Cognitive Science*, 1990: 179-211.

Fogel, David B. *Blondie 24*. San Francisco: Academic Press, 2002.

Gallagher, Marcus, and Mark Ledwich. "Evolving Pac-Man Players: Can We Learn from Raw Input." *Computational Intelligence and Games*, 2007: 282-287.

Galway, Leo, Darrly Charles, and Michaela Black. "Improving Temporal Difference Game Agent Control Using a Dynamic Explorlation Rate During Control Learning." *Proceedings of the 5th international conference on Computational Intelligence and Games*, 2009: 38-45.

Gilbert, Charles D., and Mariano Sigman. "Brain States: Top-Down Influences in Sensory Processing." *Neuron*, 2007: 677-696.

Hamid R. Maei, Csaba Szepesv´ari, Shalabh Bhatnagar, Doina Precup, David Silver, Richard S. Sutton. "Convergent Temporal Difference Learning with Arbitrary Smooth Function Approximation." *Advances in Neural Processing Systems*, 2009.

Häusser, Michael, Burt Sakmann, Nelson Spruston, and Greg. Stuart. "Action Potential Initiation and Backpropagation in Neurons of the Mammalian CNS." *Trends in Neuroscience*, 1997: 125-131.

Hawkins, Jeff. *On Intelligence.* New York: Henry Holt and Company, 2004.

Haykin, SImon. *Neural Networks and Learning Machines.* Upper Saddle River: Prentice Hall, 2009.

Hodges, Don. *Why do Pinky and Inky have different behaviors when Pac-Man is facing up?* December 30, 2008. http://donhodges.com/pacman_pinky_explanation.htm.

Jones, M. Tim. *Artificial Intelligence: A Systems Approach.* Boston: Infinity Science Press, 2008.

Kohonen, Tuevo. "Self-organized Formation of Topologically Correct Feature Map." *Biological Cybernetics*, 1982: 59-69.

Lucas, Simon M. "Evolving a neural network location evaluator to play Ms. Pac-Man." *IEEE Symposium on Computational Intelligence and Games*, 2005: 203-210.

Maei, Hamid Reza, and Richard S. Sutton. "GQ(λ): A General Gradient Algorithm for Temporal-Difference Prediction Learning With Eligibility Traces." *Proceedings of the Third Conference on Artificial General Intelligence*, 2010.

Pittman, Jamey. *The Pac-Man Dossier.* January 2009. http://home.comcast.net/~jpittman2/pacman/pacmandossier.html.

Pöppel, Erns, and Arrias-Carrión, Óscar. "Dopamine, Learning, and Reward-Seeking Behavior." *ACTA Neurobiologia Experimentalis*, 2007: 481-488.

Sutton, Richard S. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning*, 1988: 9-44.

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Boston: MIT Press, 1998.

Tesauro, Gerald. "Practical Issues in Temporal Difference Learning." *Machine Learning*, 1992: 257-277.

Tesauro, Gerald. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM*, 1995.

The Dot Eater. 2006. http://thedoteaters.com/p2_stage4.php.

Yao, Xin. "Evolving Artificial Neural Networks." *Proceedings of the IEEE*, 1999.