In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO-2006), pp. 1321-1328, Seattle, WA, July 2006

Comparing Evolutionary and Temporal Difference Methods in a Reinforcement Learning Domain

Matthew E. Taylor
mtaylor@cs.utexas.edu

Shimon Whiteson
shimon@cs.utexas.edu

Peter Stone
pstone@cs.utexas.edu

Department of Computer Sciences
The University of Texas at Austin
1 University Station, C0500
Austin, Texas 78712-1188

ABSTRACT

Both genetic algorithms (GAs) and temporal difference (TD) methods have proven effective at solving reinforcement learning (RL) problems. However, since few rigorous empirical comparisons have been conducted, there are no general guidelines describing the methods' relative strengths and weaknesses. This paper presents the results of a detailed empirical comparison between a GA and a TD method in Keepaway, a standard RL benchmark domain based on robot soccer. In particular, we compare the performance of NEAT [19], a GA that evolves neural networks, with Sarsa [16, 17], a popular TD method. The results demonstrate that NEAT can learn better policies in this task, though it requires more evaluations to do so. Additional experiments in two variations of Keepaway demonstrate that Sarsa learns better policies when the task is fully observable and NEAT learns faster when the task is deterministic. Together, these results help isolate the factors critical to the performance of each method and yield insights into their general strengths and weaknesses.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Performance, Experimentation

Keywords

Empirical study, Genetic algorithms, Machine learning, Performance analysis

1. INTRODUCTION

Reinforcement learning (RL) problems are characterized by agents making sequential decisions with the goal of maximizing total reward, which may be time delayed. RL problems contrast with classical planning problems in that agents do not know *a priori* how their actions will affect the world. RL differs from supervised learning because the agent is never given training examples with the correct action labeled.

Temporal difference (TD) [25] methods are one popular way to solve RL problems. TD methods learn a *value function* that estimates the expected long-term reward for taking a particular action in a state. *Genetic algorithms* (GAs) can also address RL problems by searching the space of policies for one that receives maximal reward. Although these two approaches have both had success in difficult RL tasks, only a few studies (e.g [4, 9, 11]) have directly compared them. As a result, there are currently no general guidelines describing the methods' relative strengths and weaknesses.

This paper presents the results of a detailed empirical comparison between a GA and a TD method. In particular, we compare the performance of *NeuroEvolution of Augmenting Topologies* (NEAT) [19] with *Sarsa* [16, 17]. NEAT, a GA that evolves neural networks, has had substantial success in RL domains like pole balancing [19], game playing [21], and robot control [20]. Sarsa is a popular TD method that has also had empirical success [23, 24, 25].

These comparisons are conducted in *3 vs. 2 Keepaway* [22], a standard RL benchmark domain based on robot soccer in which agents have noise in both their sensors and actuators. Keepaway is an appealing platform for empirical comparisons because the performance of TD methods in it has already been established in previous studies [8, 23]. While GAs have been applied to variations of Keepaway [7, 26], they have never been applied to the benchmark version of the task. We compare NEAT to Sarsa with radial basis function approximators, the best performing TD method to date [22]. Our results in this domain demonstrate that NEAT discovers better policies, though it requires many more evaluations to do so.

In addition, this paper presents the results of experiments conducted in fully observable and deterministic variations of Keepaway that are designed to isolate factors critical to the performance of each method. Results in these variations demonstrate that the performance of Sarsa is improved if the task is fully observable and the speed of NEAT is greatly improved if the fitness function is deterministic. Together, these results shed light on the open question of when GAs or TD methods perform better and why.

The remainder of this paper is organized as follows. Section 2 introduces NEAT and Sarsa. Section 3 gives details of the Keepaway domain. Section 4 shows how we apply the NEAT and Sarsa learning algorithms to this task. Section 5 presents and discusses the results of our experiments. Section 6 addresses future work and Section 7 concludes.

2. BACKGROUND

In this section we provide an overview of the GA and TD methods, NEAT and Sarsa, used in our experiments. There are a wide variety of both GAs and TD methods in use today but in order to compare these different approaches empirically we must focus on specific instantiations. We use Sarsa and NEAT as representative methods because of their empirical success in the benchmark Keepaway task [23] or variations thereof [26].

Sarsa and NEAT are also the methods that the authors are most familiar with. In addition to the obvious practical advantages, this familiarity enables us to set both algorithms' parameters with confidence. Throughout our experiments, we tried to give equal effort to optimizing the parameters of each algorithm, though such factors are admittedly difficult to control for. Wherever possible, we quantify the amount of tuning involved.

2.1 NeuroEvolution of Augmenting Topologies (NEAT)[1]

The experiments in this paper use NeuroEvolution of Augmenting Topologies (NEAT) as a representative evolutionary method for RL. NEAT is an appropriate choice because of its empirical successes on difficult RL tasks like pole balancing [19], game playing [21], and robot control [20].

In a typical neuroevolutionary system [28], the weights of a neural network are strung together to form an individual genome. A population of such genomes is then evolved by evaluating each one and selectively reproducing the fittest individuals through crossover and mutation. Most neuroevolutionary systems require the designer to manually determine the network's topology (i.e. how many hidden nodes there are and how they are connected). By contrast, NEAT automatically evolves the topology to fit the complexity of the problem. It combines the usual search for network weights with evolution of the network structure. The remainder of this section provides a brief overview of this process. Stanley and Miikkulainen [19] present a full description.

2.1.1 Minimizing Dimensionality

Unlike other systems that evolve network topologies and weights [6, 28], NEAT begins with a uniform population of simple networks with no hidden nodes and inputs connected directly to outputs. New structure is introduced incrementally via two special mutation operators. Figure 1 depicts these operators, which add new hidden nodes and links to the network. Only the structural mutations that yield performance advantages tend to survive evolution's selective pressure. In this way, NEAT tends to search through a minimal number of weight dimensions and find an appropriate complexity level for the problem.

[1]Section 2.1 is adapted from the original NEAT paper [19].

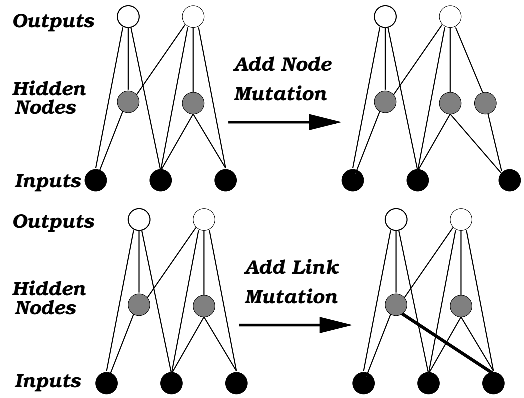

Figure 1: Examples of NEAT's mutation operators for adding structure to networks. At top, a hidden node is added by splitting a link in two. At bottom, a link, shown with a thicker black line, is added to connect two nodes.

2.1.2 Genetic Encoding with Historical Markings

Evolving network structure requires a flexible genetic encoding. Each genome in NEAT includes a list of *connection genes*, each of which refers to two *node genes* being connected. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows NEAT to find corresponding genes during crossover.

To perform crossover, the system must be able to tell which genes match up between *any* individuals in the population. For this purpose, NEAT keeps track of the historical origin of every gene. Whenever a new gene appears (through structural mutation), a *global innovation number* is incremented and assigned to that gene. The innovation numbers thus represent a chronology of every gene in the system. Whenever these genomes crossover, innovation numbers on inherited genes are preserved. Thus, the historical origin of every gene in the system is known throughout evolution.

Through innovation numbers, the system knows which genes match up with which. Genes that do not match are either *disjoint* or *excess*, depending on whether they occur within or outside the range of the other parent's innovation numbers. When crossing over, the genes in both genomes with the same innovation numbers are lined up. Genes that do not match are inherited from the more fit parent, or if they are equally fit, from both parents randomly.

Historical markings allow NEAT to perform crossover without expensive topological analysis. Genomes of different organizations and sizes stay compatible throughout evolution, and the problem of matching different topologies [15] is essentially avoided.

2.1.3 Speciation

In most cases, adding new structure to a network initially reduces its fitness. However, NEAT speciates the population every generation so that individuals compete primarily within their own niches rather than with the population at large. Hence topological innovations are protected and have time to optimize their structure before competing with other niches in the population.

Historical markings make it possible for the system to divide the population into species based on topological similarity. The distance δ between two network encodings is a simple linear combination of the number of excess (E) and disjoint (D) genes, as well as the average weight differences of matching genes ($\overline{W}$):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}. \quad (1)$$

The coefficients c_1, c_2, and c_3 adjust the importance of the three factors, and the factor N, the number of genes in the larger genome, normalizes for genome size. Genomes are tested one at a time; if a genome's distance to a randomly chosen member of the species is less than δ_t, a compatibility threshold, it is placed into this species. Each genome is placed into the first species where this condition is satisfied, so that no genome is in more than one species.

The reproduction mechanism for NEAT is *explicit fitness sharing* [3], where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

2.1.4 NEAT for Reinforcement Learning

Since NEAT is a general purpose optimization technique, it can be applied to a wide variety of problems. In this paper, we use NEAT to perform policy search RL. Each neural network in the population represents a candidate policy in the form of an *action selector*. The inputs to the network describe the agent's current state. There is one output for each available action; the agent takes whichever action has the highest activation.

A candidate policy is evaluated by allowing the corresponding network to control the agent's behavior and observing how much reward it receives. The policy's fitness is simply the sum of the rewards the agent accrues while under the network's control. In deterministic domains, each member of the population can be evaluated in a single episode. However, most real-world problems are non-deterministic and hence the reward a policy receives over the course of an episode may have substantial variance. In such domains, it is necessary to evaluate each member of the population for many episodes to get accurate fitness estimates.

2.2 Sarsa

The experiments presented in this paper use Sarsa as a representative TD method. Sarsa is an appropriate choice because of its multiple empirical successes [23, 24, 25]. Sarsa is a TD method that learns to estimate the action-value function, $Q(s, a)$, which predicts the long-term expected return of taking a particular action, a, in a particular state, s. Learning a value function allows the learner to estimate the efficacy of each action in a given state. By contrast, GAs evaluate entire policies holistically and hence have no notion of the value of individual actions.

Sarsa is an acronym for State Action Reward State Action, describing the 5-tuple needed to perform the update: (s, a, r, s', a'), where s and a are the the agent's current state and action, r is the immediate reward the agent receives from the environment, and s' and a' are the agent's subsequent state and chosen action. In the simple case, the action-value function is represented in a table, with one entry for each state-action pair. After each action, the table is updated according to the following rule:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma Q(s', a')) \quad (2)$$

where α is the learning rate and γ is a discount factor used to weight immediate rewards more heavily than future rewards.

Like other TD methods, Sarsa estimates the value of a given state-action pair by bootstrapping off estimates of other such pairs. In particular, the value of a given state-action pair (s, a) can be estimated as $r + \gamma Q(s', a')$, which is the discounted value of the subsequent state-action pair (s', a') plus the immediate reward received during the transition. Sarsa's update rule takes the old action-value estimate $Q(s, a)$, and moves it incrementally closer towards this new estimate. The learning rate parameter α controls the size of these increments. Ideally, these action-value estimates will become more accurate over time and the agent's policy will steadily improve.

In continuous domains like Keepaway, the value function cannot be represented in a table. In such cases, TD methods rely on *function approximators*, which map state-action pairs to values via more concise, parameterized functions and use supervised learning methods to set these parameters. Many function approximators have been used, including neural networks, CMACs, and radial basis functions [25]. In this paper we use a radial basis function approximator (RBF), a method with previous empirical successes [13, 22].

3. THE BENCHMARK KEEPAWAY TASK

To test the relative efficacy of NEAT and Sarsa, we use the benchmark 3 vs. 2 Keepaway task. Keepaway is an appealing platform for empirical comparisons because the performance of TD methods has already been established in previous studies [8, 23]. While GAs have been applied to variations of Keepaway [7, 26], they have never, to our knowledge, been applied to the task's benchmark version.

Keepaway is part of the open source RoboCup Soccer Server [10], and we set parameters the same as in our past research [22, 23]. RoboCup simulated soccer is well understood as it has been the basis of multiple international competitions and research challenges. This multiagent domain has noisy sensors and actuators as well as hidden state so that agents have only a partial view of the world.

In Keepaway, a subproblem of the full 11 vs. 11 simulated soccer game, a team of *keepers* attempts to maintain possession of the ball on a 20m x 20m field while one or more *takers* attempt to gain possession of the ball or force the ball out of bounds, ending an *episode*. Figure 2 depicts three keepers playing against two takers. All our experiments are run on a code base derived from version 0.6 of the benchmark Keepaway implementation[2] [22].

Three keepers are initially placed in three corners of the field and a ball is placed near one of the keepers. The two takers are placed in the fourth corner. When an episode starts, the three keepers attempt to keep control of the ball by passing among themselves and moving to open positions. The agent's state is defined by 13 variables, as shown in Figure 2. The keepers receive a reward of +1 for every time step that the ball remains in play. The episode finishes when a taker gains control of the ball or the ball is kicked out of bounds. The episode is then reset with a random keeper placed near the ball. The initial state is different in each episode because the same keeper does not always start in the same corner and because the keepers are only placed near the corners rather than in exact locations.

[2] Available at `http://www.cs.utexas.edu/~AustinVilla/sim/Keepaway/`

The agents choose not from the simulator's primitive actions but from a set of higher-level macro-actions implemented as part of the player. These macro-actions can last more than one time step and the keepers make decisions only when a macro-action terminates. The macro-actions are Hold Ball, Get Open, Receive, and Pass [23]. The agents make decisions at discrete time steps, at which point macro-actions are initiated and terminated. Takers do not learn and always follow a static hand-coded strategy.

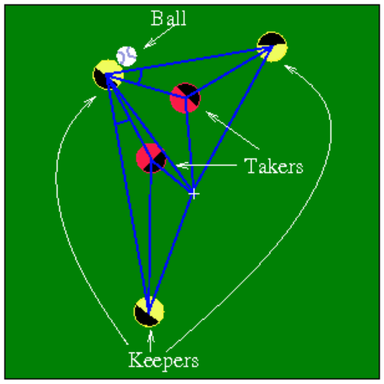

Figure 2: 13 state variables are used for learning with 3 keepers and 2 takers. The state is ego-centric and rotationally invariant for the keeper with the ball; there are 11 distances, indicated with blue lines, between players and the center of the field as well as 2 angles along passing lanes.

The keepers learn in a constrained policy space: they have the freedom to decide which action to take only when in possession of the ball. A keeper in possession may either hold the ball or pass to one of its teammates. Therefore, in 3 vs. 2 Keepaway, a keeper with the ball may choose from 3 actions, $A = \{\text{hold, passToTeammate1, passToTeammate2}\}$. Keepers not in possession of the ball are required to execute the Receive macro-action in which the keeper who can reach the ball the fastest goes to the ball and the remaining players follow a hand-coded strategy to try to get open for a pass.

In the standard Keepaway task, noise in the sensors and actuators causes the evaluation of a policy to have a large variance; the standard deviation of hold times from ten Keepaway episodes is roughly half of the mean hold time. Hence, when testing the performance of a policy after learning, we evaluate it for 1,000 episodes.

The RoboCup Soccer Server's time steps are in 0.1 second increments and all times reported in this paper refer to simulator time. Thus we only report sample complexity and not computational complexity; the running time for our learning methods is negligible compared to that of the Soccer Server. The machines used for our experiments allowed us to speed up the simulator by a factor of two so that the real experimental time required was roughly half that of the reported simulator time.

4. LEARNING IN KEEPAWAY

This section describes how we apply the NEAT and Sarsa learning algorithms to the Keepaway domain.

4.1 NEAT

Every network evolved by NEAT has 13 inputs, corresponding to the Keepaway state variables, and 3 outputs, corresponding to the available macro-actions. The keepers always select the action with the highest activation, breaking ties randomly. We found that a population of 100 organisms with a target of 5 species and the default values of $c_1 = 1.0$, $c_2 = 1.0$, and $c_3 = 2.0$ allowed NEAT to learn well in the benchmark task. See Appendix A for additional NEAT parameters used.

We used NEAT to evolve teams of homogeneous agents: in any given episode, the same neural network is used to control all three keepers on the field. The reward accrued during that episode then contributes to NEAT's estimate of that network's fitness. While heterogeneous agents could be evolved using cooperative coevolution [12], doing so is beyond the scope of this paper.

Since the Keepaway task is stochastic and the evaluations are noisy, it is difficult to establish *a priori* the optimal number of episodes to evaluate each NEAT organism. To set this parameter, we generated a number of NEAT learning curves with the number of Keepaway episodes per generation set to one of {1,000, 2,000, 6,000, 10,000} and found that 6,000 episodes per generation yielded the best performance.

Another difficult question is how to distribute these episodes among the organisms in a particular generation, given a noisy fitness function. While previous researchers have developed statistical schemes for performing such allocations [1, 18], in this paper we adopt a simple heuristic strategy to increase the performance of NEAT: we concentrate evaluations on the more promising organisms in the population because their offspring will populate the majority of the next generation. In each generation, every organism is initially evaluated for ten episodes. After that, the highest ranked organism that has not already received 100 episodes is always chosen for evaluation. Hence, every organism receives at least 10 evaluations and no more than 100, with the more promising organisms receiving the most.

4.2 Sarsa

While it is simpler in NEAT to learn homogeneous teams of agents, the opposite is true in Sarsa. Unlike NEAT, Sarsa's learning rule is applied after each action is taken. If the team is homogeneous, then each agent must update the same value function, which is infeasible in Keepaway since communication between the agents is forbidden. Hence, we use Sarsa to learn teams of heterogeneous agents, with each keeper independently updating its own value function. This setup might appear to give Sarsa a disadvantage, since learning three policies is presumably harder than learning one. However, we found that Sarsa's performance does not improve when inter-agent communication is allowed and Sarsa is used to train homogeneous teams.

In our experiments, Sarsa's parameter values were the same as in our previous work on Keepaway [22] except that we tuned the learning rate, α, to maximize the performance of the learned policies. After trying four different values, we set it to 0.05. The discount factor, γ was set to 1.0. Since learners must select from macro-level actions that may take multiple time-steps, we utilized a distributed SMDP [2, 14] version of Sarsa, as in our previous Keepaway research [23].

Given the estimated Q-values for the current state, the agent needs a method of picking an action based on these values. If the agent always behaves greedily and selects the action with the highest estimated value, it will never explore potentially superior alternatives. To improve its policy, the agent must occasionally try other actions. We utilize the standard ϵ-greedy action selection mechanism to ensure that this occurs. With probability $1-\epsilon$, the agent takes the greedy action, while with probability ϵ it selects a random action. Our experiments set ϵ to 0.01, as was done previously [22]. We also tried different values of ϵ and allowed it to decay over time at different rates but did not find noticeable improvements in learning performance.

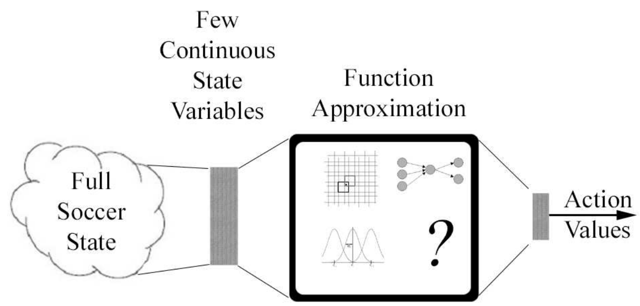

Figure 3: Function approximation is necessary for agents learning in a continuous world. This diagram schematically depicts how a TD agent produces state-action values. In this paper we use RBF function approximation although many other function approximators have also been shown to work well with Sarsa.

We utilized a radial basis function approximator (RBF) [25] as a previous study showed that it was superior to CMAC and neural network approximators in the Keepaway domain [22] (see Figure 3 for a visualization of function approximation in Keepaway). When considering a single state variable, an RBF approximator is a linear function approximator

$$\hat{f}(x) = \sum_i w_i f_i(x) \tag{3}$$

where the basis functions have the form

$$f_i(x) = \phi(|x - c_i|) \tag{4}$$

where x is the current state, c_i is the center of feature i, and w_i represents weights that can be modified over time by a learning algorithm. Here we set the features to be evenly spaced Gaussian radial basis functions, where

$$\phi(|x - c_i|) = \exp(-\frac{|x - c_i|^2}{2\sigma^2}) \tag{5}$$

(see Figure 4). The σ parameter controls the width of the Gaussian function and therefore the amount of generalization over the state space. We set σ to 0.25, as in previous studies [22].

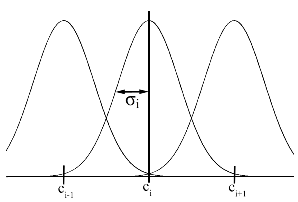

Figure 4: An RBF approximator computes Q(s,a) via a weighted sum of Gaussian functions. The contribution from the i^{th} Gaussian is weighted by the distance from its center, c_i, to the relevant state variable. σ can be tuned to control the width of Gaussians and thus how much the function approximator generalizes.

A state s is composed of some number n of state variables, $s_0 \ldots s_{n-1}$. We assumed that the state variables are independent as was done previously [22, 23] and thus have one set of linearly tiled RBFs for each state variable. We use equations 3-5 to calculate Q-values of a state s, once for each action, where $Q(s, a) = \hat{f}(x)$, $s = x$, and there are n sets of tilings (one for each state variable). All weights w_i are initially set to zero, but over time Sarsa changes the values of the weights so that the resulting Q-values more closely predicted the true returns, as specified by equation 2.

5. RESULTS AND DISCUSSION

This section presents results comparing the performance of NEAT and Sarsa in the benchmark Keepaway task. We also present results in two variations of Keepaway designed to isolate factors critical to each method's performance.

In order to perform such comparisons, we need a way to measure the quality and speed of learning for each method. In other words, we need to measure the quality of the best policy each method has discovered so far at various points in the learning process. For Sarsa, this is just the greedy policy ($\epsilon = 0.0$) that corresponds to the agent's current estimate of the value function. For NEAT, it is the champion of the most recently completed generation.

Since Keepaway evaluations are noisy and Sarsa uses exploration ($\epsilon \neq 0.0$) while learning, the quality of the best policy at a given point cannot be definitely established from each method's performance during learning. Instead, we assess the policies in retrospect by conducting additional evaluations after the learning runs have completed.

For NEAT, we take the champion from each generation and evaluate it for 1,000 episodes. For Sarsa, we take the estimated value function at 1,000 episode intervals and evaluate the corresponding greedy policy for 1,000 episodes.

5.1 Benchmark Keepaway Results

To compare the performance of NEAT and Sarsa in the benchmark Keepaway task, we conducted a series of independent trials of each method. We ran each trial until it plateaued, i.e. its performance did not improve for several hours. Doing so enabled us to generate more data with fixed computational resources. Since Sarsa trials plateaued much sooner than NEAT trials (89 hours versus 840 hours[3] of simulator time, on average), we were able to conduct a total of 20 Sarsa runs but only 5 NEAT runs.

As mentioned above, we computed the mean hold time of the best policy found so far by each method at regular intervals. For each trial, these means were computed by averaging performance over 1,000 test episodes. The mean hold times were then averaged across all trials of each of the two methods to obtain the plots shown in Figure 5. Note that because the Sarsa learning curves plateau before the NEAT learning curves, the performance of the Sarsa learners is extended on the graph even after learning has finished, denoted by a horizontal performance line without plotted data points. For presentation purposes we plot the average performance every 10 hours for the first 200 hours and then every 50 hours after that. Increasing the sampling resolution does not reveal any interesting detail in the learning curves. We plot the performance at time 0 only for Sarsa learning curves. The Sarsa policy can be evaluated before training but, because NEAT must spend a number of episodes testing each organism in the population before determining a champion, at time 0 there is no champion to evaluate.

The results demonstrate that NEAT can learn better policies in the benchmark Keepaway task than Sarsa with RBFs, the best performing TD method to date. A Student's t-test confirms that the difference in performance between NEAT and Sarsa is statistically significant for times greater than or equal to 650 hours ($p < 0.014$). These results also highlight an important trade-off between the two methods

[3]For reference, 840 hours of simulator time in the benchmark Keepaway task corresponded to roughly 57 generations, 342,000 episodes, or 420 hours of real time.

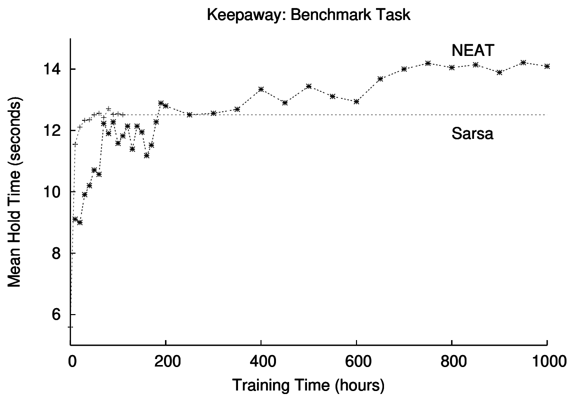

Figure 5: A comparison of the mean hold times of the policies discovered by NEAT and Sarsa in the benchmark Keepaway task.

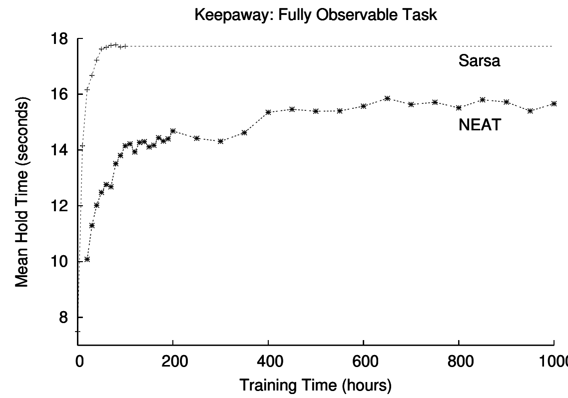

Figure 6: A comparison of the mean hold times of the policies discovered by NEAT and Sarsa in the fully observable version of the Keepaway task.

tested. While NEAT ultimately learns better policies, it requires many more evaluations to do so. Sarsa learns much more rapidly: in the early part of learning its average policy is much better than NEAT's.

In order to discover what characteristics of the Keepaway task most critically affect the speed and quality of learning in each method, we conducted additional experiments in two variations of Keepaway. The remainder of this section describes these variations and the results of our experiments therein. We found that making the Keepaway task fully observable benefits Sarsa relative to NEAT because one of the fundamental TD assumptions that yields convergence guarantees is no longer violated. By contrast, when the Keepaway task is made fully deterministic and fully observable, NEAT benefits relative to Sarsa because the number of evaluations each generation requires for successful learning is dramatically reduced.

5.2 Fully Observable Keepaway Results

In the benchmark Keepaway task, the agents' sensors are noisy. Since the agents can only partially observe the true state of the world, the task is non-Markovian, i.e. the probability distribution over next states is not independent of the agents' state and action histories. This fact could be problematic for Sarsa since the principles underlying TD update rules assume the environment is Markovian, though in practice they can still perform well when it is not [25]. By contrast, NEAT can evolve recurrent networks that cope with non-Markovian tasks by recording important information about previous states [5]. NEAT makes use of this ability in the benchmark task as the champion organisms typically contain recurrent links. Therefore, we hypothesized that Sarsa's relative performance would improve if sensor noise was removed, rendering the Keepaway task fully observable and effectively Markovian.[4]

[4]The state is not truly Markovian because player velocities are not included. If the agent stored past states it could calculate these velocities and therefore better predict future states. However, the Keepaway benchmark task does not include velocity because past research did not find it useful for learning; players have low inertia and the field has a high coefficient of friction which means that velocity does not help agents learn in practice. In this paper we use the same state variables as previous work [22, 23] but note that when sensor noise is removed the state is "effectively Markovian."

To test this hypothesis we conducted 5 trials of NEAT and 20 trials of Sarsa in the fully observable Keepaway task. Figure 6 shows the results of these experiments, with mean hold times computed as before and averaged across all trials of a single method. As in the benchmark version of the task, Sarsa learns much more rapidly than NEAT. However, in the fully observable version, Sarsa also learns substantially better policies. The difference in performance between NEAT and Sarsa is statistically significant for all points graphed ($p < 1.0 \times 10^{-4}$).

These results confirm our hypothesis that full observability is a critical factor in Sarsa's performance in the Keepaway task. While Sarsa can learn well in the partially observable benchmark version of the task, its performance relative to NEAT improves dramatically when sensor noise is removed. These results are not surprising given the way these two methods work: Sarsa's underlying assumptions are violated in the absence of the Markov property. By contrast, NEAT makes no such assumption and therefore tasks with partial observability are not particularly problematic.

5.3 Deterministic Keepaway Results

Even in the fully observable version of the task, which has no sensor noise, the Keepaway task remains highly stochastic due to noise in the agents' actuators and randomness in the agents' initial states. In both the benchmark task and the fully observable variation described above, this stochasticity greatly slows NEAT's learning rate. A policy's measured fitness in a given episode has substantial variance and NEAT is able to learn well only when the fitness estimate for each candidate policy is averaged over many episodes. Therefore, we tested a second variation of Keepaway in which noise was removed from the sensors and the actuators and the agent's initial states were fixed. These changes yield a domain with a completely deterministic fitness function.[5] We hypothesized that NEAT would learn much faster in this deterministic variation as it could perfectly evaluate each organism in a single episode.

[5]There is also a third possible variation of Keepaway which is partially observable but does not have stochastic actions or random initial states. We do not study this variation in this paper because it is not fundamentally different from the benchmark task: it is still non-Markovian and still has a noisy fitness function. Informal experiments suggest that in this version of the task the relative performance of NEAT and Sarsa are unchanged from the benchmark task.

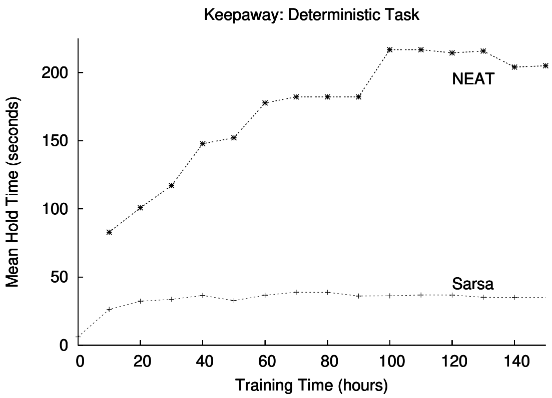

Figure 7: A comparison of the mean hold times of the policies discovered by NEAT and Sarsa in the deterministic version of the Keepaway task.

To test this hypothesis, we conducted 5 trials of NEAT and 20 trials of Sarsa in the deterministic Keepaway task. These experiments used the same parameters as those described above except NEAT conducted only 100 episodes per generation instead of 6,000 because each of the 100 organisms in the population required only 1 evaluation episode. We hypothesized that the Sarsa learners would benefit from an increased learning rate in the fully deterministic environment and experimented with different learning rates, but found no improvement over the original value of 0.05. Figure 7 shows the results of these experiments, with mean hold times computed as before and averaged across all trials of a single method. In the deterministic version of the task, Sarsa's speed advantage disappears. NEAT learns more rapidly than Sarsa, in addition to discovering dramatically superior policies. The difference in performance between NEAT and Sarsa is statistically significant for all points graphed ($p < 2.6 \times 10^{-6}$).

These results confirm our hypothesis that stochasticity in the Keepaway domain critically affects how quickly NEAT can learn. In the benchmark task, NEAT learns well only when fitness estimates are averaged over many episodes, resulting in slow learning relative to Sarsa. By contrast, in the deterministic version, only one episode of evaluation is necessary for each organism, enabling NEAT to learn much more rapidly. Furthermore, making the Keepaway task deterministic greatly improves, relative to Sarsa, the quality of the best policies discovered. This outcome is surprising since the deterministic version of the task is also fully observable and should be well suited to TD methods. These results demonstrate that in the deterministic version of the task the advantage Sarsa gains from full observability is far outweighed by the advantage NEAT gains from the ability to perform rapid and accurate fitness evaluations.

Our results therefore suggest that the choice between using GAs and TD methods can be made based on some of the target task's characteristics. In deterministic domains where the fitness of an organism can be quickly evaluated, GAs are likely to excel. If the task is fully observable but non-deterministic, TD methods may have a critical advantage. If the task is partially observable and non-deterministic, each method may have different advantages: TD methods in speed and GAs in ultimate performance.

6. FUTURE WORK

As stated in Section 2, no comparison between two learning methods that are parameterized differently can be completely objective. Most learning methods have some number of parameters to set, and the amount of time devoted to "tweaking" them can have a dramatic impact on the success of learning. Furthermore, no empirical study is able to compare all algorithmic variants or consider every relevant domain. Though both the algorithms and the domain used in this paper were carefully chosen to be representative, further data is clearly needed to make conclusive statements about the relative strengths and weaknesses of GAs and TD methods. We hope that in the future there will be many more studies that address these issues.

Another open question is how to integrate GAs and TD methods such that their contrasting strengths can be exploited simultaneously. One promising approach is evolutionary function approximation [27], which uses GAs to optimize TD function approximators. This and other hybrid approaches may perform well in Keepaway tasks. Extending our comparison to include such methods would likely enhance our understanding of the differences between GAs and TD methods.

7. CONCLUSION

This paper presents the results of a detailed empirical comparison between NEAT and Sarsa in Keepaway, a standard RL benchmark domain based on robot soccer. The results demonstrate that NEAT can learn better policies in this domain than Sarsa, the previous best known method, though it requires more evaluations to do so. Additional experiments in two variations of Keepaway demonstrate that Sarsa learns better policies when the domain is fully observable and NEAT learns faster when the domain has a deterministic fitness function. Together, these results help isolate factors critical to the performance of each method and yield insights into how they may perform on additional tasks. Additional studies using more domains and different algorithms are necessary to draw definitive guidelines about when to use a GA or TD method. This study provides a model of how comparisons may be done fairly and provides an initial data point for establishing such guidelines.

Acknowledgments

We would like to thank Ken Stanley for help setting up NEAT in Keepaway, as well as Nate Kohl, David Pardoe, Joeseph Reisinger, Jefferson Provost, and the anonymous reviewers for helpful comments and suggestions. This research was supported in part by DARPA grant HR0011-04-1-0035, NSF CAREER award IIS-0237699, and NSF award EIA-0303609.

APPENDIX

A. NEAT PARAMETERS

This section details the NEAT parameters used in our experiments. Stanley and Miikkulainen [19] describe the semantics of these parameters in detail. The coefficients for measuring compatibility were $c_1 = 1.0$, $c_2 = 1.0$, and $c_3 = 2.0$. The compatibility distance δ_t was adjusted dynamically to maintain a target of 5 species. The survival threshold was 0.2. The weight mutation power was 0.01. The interspecies mating rate was 0.05. The drop-off age was 1,000. The probability of adding recurrent links was 0.2.

B. REFERENCES

[1] T. Beielstein and S. Markon. Threshold selection, hypothesis tests and DOE methods. In *2002 Congresss on Evolutionary Computation*, pages 777–782, 2002.

[2] S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 393–400, San Mateo, CA, 1995. Morgan Kaufmann.

[3] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154, 1987.

[4] F. Gomez and R. Miikkulainen. Robust nonlinear control through neuroevolution. Technical Report AI02-292, Department of Computer Sciences, University of Texas at Austin, 2002.

[5] F. Gomez and J. Schmidhuber. Co-evolving recurrent neurons learn deep memory POMDPs. In *GECCO-05: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 491–498, 2005.

[6] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89. MIT Press, 1996.

[7] W. H. Hsu and S. M. Gustafson. Genetic programming and multi-agent layered learning by reinforcements. In *Genetic and Evolutionary Computation Conference*, pages 764–771, New York,NY, July 2002. Morgan Kaufmann.

[8] K. Kostiadis and H. Hu. KaBaGe-RL: Kanerva-based generalisation and reinforcement learning for possession football. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*, October 2001.

[9] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.

[10] I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.

[11] J. B. Pollack, A. D. Blair, and M. Land. Coevolution of a backgammon player. In C. G. Langton and K. Shimohara, editors, *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems*, pages 92–98, Cambridge, MA, 1997. The MIT Press.

[12] M. A. Potter and K. A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8:1–29, 2000.

[13] M. J. D. Powell. Radial basis functions for multivariate interpolation: A review. In J. C. Mason and M. G. Cox, editors, *Algorithms for Approximation*, pages 143–167, Oxford, 1987. Clarendon Press.

[14] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[15] N. J. Radcliffe. Genetic set recombination and its application to neural network topology optimization. *Neural computing and applications*, 1(1):67–90, 1993.

[16] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG-RT 116, Engineering Department, Cambridge University, 1994.

[17] S. P. Singh and R. S. Sutton. Reinforcement learning with replaceing eligibility traces. *Machine Learning*, 22:123–158, 1996.

[18] P. Stagge. Averaging efficiently in the presense of noise. In *PPSN V: Proceedings of the 5th Internation Conference on Parallel Problem Solving from Nature*, pages 188–200, London, UK, 1998. Springer-Verlag.

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

[20] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[21] K. O. Stanley and R. Miikkulainen. Evolving a roving eye for go. In *Proceedinngs of the Genetic and Evolutionary Computation Conference*, 2004.

[22] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu. Keepaway soccer: From machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld, and Y. Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX*. Springer Verlag, Berlin, 2006. To appear.

[23] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[24] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044, 1996.

[25] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[26] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59(1):5–30, 2005.

[27] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 2006. To appear.

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