In *Proceedings of the Genetic and Evolutionary Computation Conference Graduate Student Workshop* (GECCO 06), Seattle, WA, July 2006.

Evolutionary Function Approximation for Reinforcement Learning

Shimon Whiteson
Department of Computer Sciences
University of Texas at Austin
1 University Station, C0500
Austin, TX 78712-0233
shimon@cs.utexas.edu

ABSTRACT

Temporal difference methods are theoretically grounded and empirically effective methods for addressing reinforcement learning problems. In most real-world reinforcement learning tasks, TD methods require a function approximator to represent the value function. However, using function approximators requires manually making crucial representational decisions. This thesis investigates *evolutionary function approximation*, a novel approach to automatically selecting function approximator representations that enable efficient individual learning. This method *evolves* individuals that are better able to *learn*. I present a fully implemented instantiation of evolutionary function approximation which combines NEAT, a neuroevolutionary optimization technique, with Q-learning, a popular TD method. The resulting NEAT+Q algorithm automatically discovers effective representations for neural network function approximators. This thesis also presents *on-line evolutionary computation*, which improves the on-line performance of evolutionary computation by borrowing selection mechanisms used in TD methods to choose individual actions and using them in evolutionary computation to select policies for evaluation. I evaluate these contributions with extended empirical studies in two domains: 1) the mountain car task, a standard reinforcement learning benchmark on which neural network function approximators have previously performed poorly and 2) server job scheduling, a large probabilistic domain drawn from the field of autonomic computing. The results demonstrate that evolutionary function approximation can significantly improve the performance of TD methods and on-line evolutionary computation can significantly improve evolutionary methods.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

evolutionary computation, reinforcement learning, temporal difference methods, neural networks, on-line learning

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

1. INTRODUCTION

In many machine learning problems, an agent must learn a *policy* for selecting actions based on its current *state*. *Reinforcement learning* problems are the subset of these tasks in which the agent never sees examples of correct behavior. Instead, it receives only positive and negative rewards for the actions it tries. Since many practical, real world problems (such as robot control, game playing, and system optimization) fall in this category, developing effective reinforcement learning algorithms is critical to the progress of artificial intelligence.

The most common approach to reinforcement learning relies on the concept of *value functions*, which indicate, for a particular policy, the long-term value of a given state or state-action pair. *Temporal difference methods* (TD) [23], which combine principles of dynamic programming with statistical sampling, use the immediate rewards received by the agent to incrementally improve both the agent's policy and the estimated value function for that policy. Hence, TD methods enable an agent to learn during its "lifetime" i.e. from its individual experience interacting with the environment.

For small problems, the value function can be represented as a table. However, the large, probabilistic domains which arise in the real-world usually require coupling TD methods with a *function approximator*, which represents the mapping from state-action pairs to values via a more concise, parameterized function and uses supervised learning methods to set its parameters. Many different methods of function approximation have been used successfully, including CMACs, radial basis functions, and neural networks [25]. However, using function approximators requires making crucial representational decisions (e.g. the number of hidden units and initial weights of a neural network). Poor design choices can result in estimates that diverge from the optimal value function [2] and agents that perform poorly. Even for reinforcement learning algorithms with guaranteed convergence [3, 15], achieving high performance in practice requires finding an appropriate representation for the function approximator. As Lagoudakis and Parr observe, "The crucial factor for a successful approximate algorithm is the choice of the parametric approximation architecture(s) and the choice of the projection (parameter adjustment) method." [15, p. 1111] Nonetheless, representational choices are typically made manually, based only on the designer's intuition.

My goal is to automate the search for effective representations by employing sophisticated optimization techniques.

In this thesis, I focus on using evolutionary methods [9] because of their demonstrated ability to discover effective representations [11, 21]. Synthesizing evolutionary and TD methods results in a new approach called *evolutionary function approximation*, which automatically selects function approximator representations that enable efficient individual learning. Thus, this method *evolves* individuals that are better able to *learn*. This biologically intuitive combination has been applied to computational systems in the past [1, 5, 8, 10, 12, 18] but never, to my knowledge, to aid the discovery of good TD function approximators.

This thesis uses NeuroEvolution of Augmenting Topologies (NEAT) [21] to select neural network function approximators for Q-learning [27], a popular TD method. The resulting algorithm, called NEAT+Q, uses NEAT to evolve topologies and initial weights of neural networks that are better able to learn, via backpropagation, to represent the value estimates provided by Q-learning.

Evolutionary computation is typically applied to *off-line* scenarios, where the only goal is to discover a good policy as quickly as possible. By contrast, TD methods are typically applied to *on-line* scenarios, in which the agent tries to learn a good policy quickly *and* to maximize the reward it obtains while doing so. Hence, for evolutionary function approximation to achieve its full potential, the underlying evolutionary method needs to work well on-line.

TD methods excel on-line because they are typically combined with action selection mechanisms like ϵ-greedy selection [25]. These mechanisms improve on-line performance by explicitly balancing two competing objectives: 1) searching for better policies (*exploration*) and 2) gathering as much reward as possible (*exploitation*). This thesis investigates a novel approach called *on-line evolutionary computation*, in which selection mechanisms commonly used by TD methods to choose individual actions are used in evolutionary computation to choose policies for evaluation. I present three implementations, based on ϵ-greedy selection, softmax selection, and interval estimation, that distribute evaluations within a generation so as to favor more promising individuals.

I evaluate these contributions with extended empirical studies in two domains: 1) mountain car and 2) server job scheduling. Using these domains, my experiments test Q-learning with a series of manually designed neural networks and compare the results to NEAT+Q and regular NEAT (which trains action selectors in lieu of value functions). The results demonstrate that evolutionary function approximation can significantly improve the performance of TD methods. Furthermore, I test NEAT with and without ϵ-greedy, softmax, and interval estimation versions of evolutionary computation. These experiments confirm that these techniques can significantly improve the on-line performance of evolutionary computation.

2. BACKGROUND

I begin by reviewing Q-learning and NEAT, the algorithms that form the building blocks of our implementation of evolutionary function approximation.

2.1 Q-Learning

The experiments presented in this thesis use Q-learning because it is a well-established, canonical TD method that has also enjoyed empirical success [27, 6]. Like many other TD methods, Q-learning attempts to learn a value function $Q(s, a)$ that maps state-action pairs to values. In the tabular case, the algorithm uses the following update rule, applied each time the agent transitions from state s to state s':

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

where $\alpha \in [0,1]$ is a learning rate parameter, $\gamma \in [0,1]$ is a discount factor, and r is the immediate reward the agent receives upon taking action a. Q-learning is an *off-policy* learning method, i.e. it can learn the optimal value function regardless of what policy the agent is following, so long as there is sufficient exploration.

In domains with large or continuous state spaces, the value function cannot be represented in a table. Instead, Q-learning is coupled with a function approximator that maps state-action pairs to values via a concise, parameterized function. Many different methods of function approximation have been used successfully, including CMACs, radial basis functions, and neural networks [25]. In this thesis, I use neural network function approximators because they have proven successful on difficult reinforcement learning tasks [6, 26]. The inputs to the network describe the agent's current state; the outputs, one for each action, represent the agent's current estimate of the value of the associated state-action pairs. The initial weights of the network are drawn from a Gaussian distribution with mean 0.0 and standard deviation σ. After each action, the weights of the neural network are adjusted using backpropagation [19] such that its output better matches the current value estimate for the state-action pair: $r + \gamma \max_{a'} Q(s',a')$.

2.2 NEAT

The implementation of evolutionary function approximation presented in this thesis relies on NeuroEvolution of Augmenting Topologies (NEAT) to automate the search for appropriate topologies and initial weights of neural network function approximators. NEAT is an appropriate choice because of its empirical successes on difficult reinforcement learning tasks like pole balancing [21] and robot control [22]. In addition, NEAT is appealing because, unlike many other optimization techniques, it automatically learns an appropriate representation for the solution.

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.

Unlike other systems that evolve network topologies and weights, NEAT begins with a uniform population of simple networks with no hidden nodes and inputs connected directly to outputs. Two special mutation operators introduce new structure incrementally. Figure 1 depicts these operators, which add hidden nodes and links to the network. Only those structural mutations that improve performance tend to survive; in this way, NEAT searches through a minimal number of weight dimensions and finds the appropriate level of complexity for the problem.

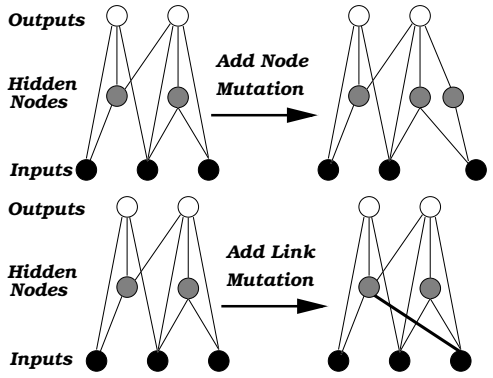


Figure 1: **Examples of NEAT's structural mutation operators. At top, a new hidden node, shown on the right, is added to the network by splitting an existing link in two. At bottom, a new link, shown with a thicker black line, is added to connect two existing nodes.**

These structural mutations result in populations of networks with varying size and shape. Mating these heterogeneous topologies requires a mechanism for deciding which genes correspond to each other. To this end, NEAT uses *innovation numbers* to track the historical origin of each structural mutation. When new genomes are created, the genes in both parents with the same innovation number are lined up; genes that do not match are inherited from the fitter parent.

Since NEAT is a general purpose optimization technique, it can be applied to a wide variety of problems. When applied to reinforcement learning problems, NEAT typically evolves *action selectors*, which directly map states to the action the agent should take in that state. Since it does not estimate value functions, it is an example of *policy search* reinforcement learning. Like other policy search methods, e.g. [14, 24], it uses global optimization techniques to directly search the space of potential policies. In the following section I describe how NEAT can be used to evolve Q-learning function approximators instead of action selectors.

3. EVOLUTIONARY FUNCTION APPROXIMATION

As described above, when evolutionary methods are applied to reinforcement learning problems, they typically evolve a population of action selectors, each of which remains fixed during its fitness evaluation. The central insight behind evolutionary function approximation is that, if evolution is directed to evolve value functions instead, then those value functions can be updated, using TD methods, during each fitness evaluation. In this way, the system can *evolve* function approximators that are better able to *learn* via TD. In addition to automating the search for effective representations, evolutionary function approximation can enable synergistic effects between evolution and learning via a biological phenomenon called the *Baldwin Effect* [4], which can speed up evolutionary computation [1, 12]. When each individual can learn during its lifetime, it need not be perfect at birth. Hence, the Baldwin Effect predicts that evolution will find good solutions more easily. In the remainder of this section, I describe NEAT+Q, a particular implementation of evolutionary function approximation.

3.1 NEAT+Q

All that is required to make NEAT optimize value functions instead of action selectors is a reinterpretation of its output values. The structure of neural network action selectors (one input for each state feature and one output for each action) is already identical to that of Q-learning function approximators. Therefore, if the weights of the networks NEAT evolves are updated during their fitness evaluations using Q-learning and backpropagation, they will effectively evolve value functions instead of action selectors. Hence, the outputs are no longer arbitrary values; they represent the long-term discounted values of the associated state-action pairs and are used, not just to select the most desirable action, but to update the estimates of other state-action pairs.

Algorithm 1 NEAT+Q$(S, A, p, m_n, m_l, g, e, \alpha, \gamma, \lambda, \epsilon)$

```
1:  // S: set of all states, A: set of all actions, p: population size
2:  // m_n: node mutation rate, m_l: link mutation rate, g: number
    of generations
3:  // e: number of episodes per generation, α: learning rate, γ:
    discount factor
4:  // λ: eligibility decay rate, ε: exploration rate
5:
6:  P[] ← INIT-POPULATION(S, A, p)
7:  for i ← 1 to g do
8:    for j ← 1 to e do
9:      N, s, s' ← P[j % p], null, INIT-STATE(S)
10:     repeat
11:       Q[] ← EVAL-NET(N, s')
12:       with-prob(ε) a' ← RANDOM(A)
13:       else a' ← argmax_k Q[k]
14:       if s ≠ null then
15:         BACKPROP(N, s, a, r + γ max_k Q[k], α, γ, λ)
16:       s, a ← s', a'
17:       r, s' ← TAKE-ACTION(a')
18:       N.fitness ← N.fitness + r
19:     until TERMINAL-STATE?(s)
20:     N.episodes ← N.episodes + 1
21:   P'[] ← new array of size p
22:   for j ←1 to p do
23:     P'[j] ← BREED-NET(P[])
24:     with-prob m_n: ADD-NODE-MUTATION(P'[j])
25:     with-prob m_l: ADD-LINK-MUTATION(P'[j])
26:   P[] ← P'[]
```

Algorithm 1 summarizes the resulting NEAT+Q method. Each time the agent takes an action, the network being evaluated is backpropagated once towards Q-learning targets (line 15) and the agent uses ϵ-greedy selection [25] to ensure it occasionally tests alternatives to its current policy (lines 12–13). If α and ϵ are set to zero, this method degenerates to regular NEAT. NEAT+Q maintains a running total of the reward accrued by the network during its evaluation (line 18). Each generation ends after e episodes, at which point each network's average fitness is $N.fitness/N.episodes$. NEAT creates a new population by repeatedly calling the BREED-NET function (line 23), which performs crossover on two highly fit parents. The new resulting network can then undergo mutations that add nodes or links to its structure (lines 24–25).

NEAT+Q combines the power of TD methods with the ability of NEAT to learn effective representations. Traditional neural network function approximators put all their eggs in one basket by relying on a single manually designed network to represent the value function. NEAT+Q, by contrast, explores the space of such networks to increase the chance of finding a representation that will perform well.

3.2 Results

As an initial baseline, I conducted 25 runs in each domain in which NEAT attempts to discover good action selectors. Next, I performed 25 runs in each domain using NEAT+Q. To test Q-learning without NEAT, I tried 24 different configurations in each domain. For simplicity, the graphs that follow show results from only the highest performing Q-learning configuration.

Figure 2 shows the results of these experiments. For each method, the corresponding line in the graph represents a uniform moving average over the aggregate utility received in the past 1,000 episodes, averaged over all 25 runs. Even though NEAT and NEAT+Q have populations instead of single networks, they used exactly the same number of episodes in training as Q-learning and hence the comparison is fair. These graphs show the average reward received during those episodes and therefore reflect performance of the entire population, not just the generation champions. Error bars indicate 95% confidence intervals. In addition, Student's t-tests confirmed, with 95% confidence, the statistical significance of the performance difference between each pair of methods.

Note that the progress of NEAT+Q consists of a series of 10,000-episode intervals. Each of these intervals corresponds to one generation and the changes within them are due to learning via Q-learning and backpropagation. Though each individual learns for 100 episodes, those episodes do not occur consecutively but are spread across the entire generation. Hence, each individual changes gradually during the generation as it is repeatedly evaluated. The result is a series of intra-generational learning curves within the larger learning curve.

For the particular problems tested and network configurations tried, evolutionary function approximation significantly improves performance over manually designed networks. Nonetheless, additional engineering of the network structure and initial weights could in principle significantly improve Q-learning's performance. I verified this fact by starting Q-learning with the best networks discovered by NEAT+Q and annealing the learning rate aggressively. In this scenario, Q-learning matched NEAT+Q's performance without directly using evolutionary computation. However, it is unlikely in practice that a manual search, no matter how extensive, would discover these successful topologies, which contain irregular and partially connected hidden layers.

NEAT+Q also significantly outperforms regular NEAT in both domains. In the mountain car domain, NEAT+Q learns faster, achieving better performance in earlier generations, though they plateau at nearly the same level. In the server job scheduling domain, NEAT+Q learns more rapidly and also converges to substantially higher performance. This result highlights the value of TD methods on challenging reinforcement learning problems. Even when NEAT is employed to find effective representations, the best performance is achieved only when TD methods are used to estimate a value function. Hence, the relatively poor performance of Q-learning is not due to some weakness in the TD methodology but merely to the failure to find a good representation.

Furthermore, in the scheduling domain, the advantage of NEAT+Q over NEAT is not directly explained just by the learning that occurs via backpropagation within each generation. After 300,000 episodes, NEAT+Q clearly performs better even at the beginning of each generation, before such learning has occurred. Just as predicted by the Baldwin Effect, evolution proceeds more quickly in NEAT+Q because the weight changes made by backpropagation, in addition to improving that individual's performance, alter selective pressures and more rapidly guide evolution to useful regions of the search space.

4. ON-LINE EVOLUTIONARY COMPUTATION

If e is the total number of episodes conducted in each generation and $|P|$ is the size of the population, evolutionary methods typically evaluate each member of the population for $e/|P|$ episodes. In on-line scenarios, this strategy is grossly suboptimal because it makes no attempt to properly balance exploration and exploitation within a generation. In fact, this strategy is purely exploratory, as every individual is evaluated for exactly the same number of episodes.

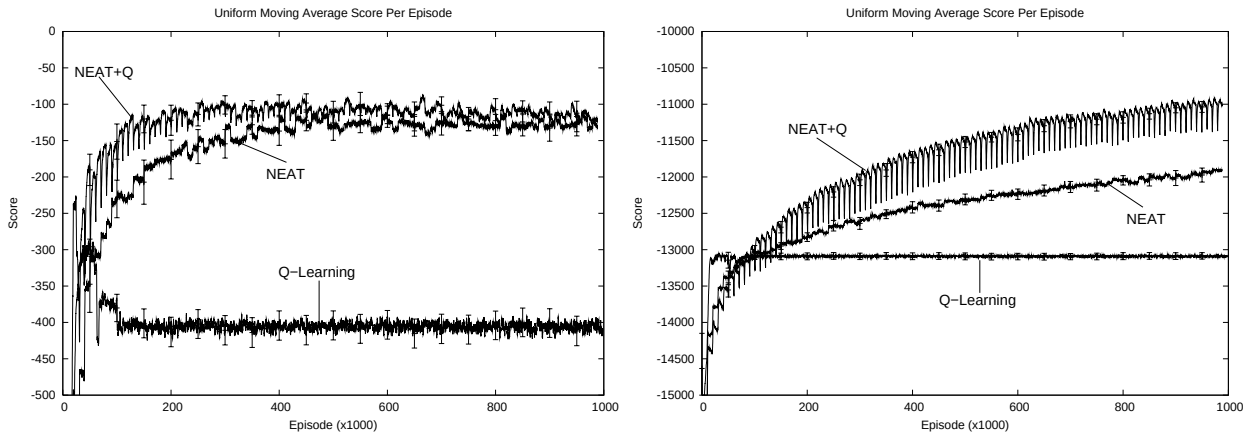
In this section, I present three methods that attempt to boost evolution's on-line performance by balancing exploration with exploitation. Instead of giving each individual the same number of episodes, these methods exploit the information gained from early episodes to favor the most promising candidate policies and thereby boost the reward accrued during learning. All three methods work by borrowing action selection mechanisms traditionally used in TD methods and applying them in evolutionary computation. In TD methods, these mechanisms directly balance exploration and exploitation by determining how often the agent behaves greedily with respect to current value estimates and how often it tries alternative actions.

In a sense, the problem faced by evolutionary methods is the opposite of that faced by TD methods. Within each generation, evolutionary methods naturally explore, by evaluating each member of the population equally, and so need a way to force more exploitation. By contrast, TD methods naturally exploit, by following the greedy policy, and so need a way to force more exploration. Nonetheless, the ultimate goal is the same: a proper balance between the two extremes.

To apply TD selection mechanisms in evolutionary computation, we must modify the level at which selection is performed. Evolutionary algorithms cannot perform selection at the level of individual actions because, lacking value functions, they have no notion of the value of individual actions. However, they can perform selection at the level of episodes, in which entire policies are assessed holistically. The same selection mechanisms used to choose individual actions in TD methods can be used to select policies for evaluation, allowing evolutionary algorithms to excel on-line by balancing exploration and exploitation within and across generations. The rest of this section details three ways to perform on-line evolution.

4.1 ϵ-Greedy Evolution

When ϵ-greedy selection is used in TD methods, a single parameter ϵ controls what fraction of the time the agent deviates from greedy behavior. Each time the agent selects an action, it chooses probabilistically between exploration and exploitation. With probability ϵ, it explores by selecting randomly from the available actions. With probability $1-\epsilon$, it exploits by selecting the greedy action.



(a) Mountain Car (b) Server Job Scheduling

Figure 2: A comparison of the performance of manual and evolutionary function approximators in the mountain car and server job scheduling domains.

In evolutionary computation, this same mechanism can be used at the beginning of each episode to select a policy for evaluation. With probability ϵ, the algorithm selects a policy randomly. With probability $1-\epsilon$, the algorithm exploits by selecting the best policy discovered so far in the current generation. The score of each policy is just the average reward per episode it has received so far. Each time a policy is selected for evaluation, the total reward it receives is incorporated into that average, which can cause it to gain or lose the rank of best policy.

To apply ϵ-greedy selection to NEAT, we need only alter the way networks are selected for evaluation. Instead of iterating through the population repeatedly until e episodes are complete, NEAT selects for evaluation, at the beginning of each episode, the policy returned by the ϵ-greedy selection function described in Algorithm 2. This function returns a policy p which is either selected randomly or which so far has the highest average fitness, $f(p)$.

Algorithm 2 ϵ-GREEDY SELECTION(P, ϵ)

```
// P: population
// ε: NEAT's exploration rate

with-prob(ε) return RANDOM(P)
else return argmax_{p∈P} f(p)
```

Using ϵ-greedy selection in evolutionary computation allows it to thrive in on-line scenarios by balancing exploration and exploitation. For the most part, this method does not alter evolution's search but simply interleaves it with exploitative episodes that increase average reward during learning. The next section describes how softmax selection can be applied to evolution to create a more nuanced balance between exploration and exploitation.

4.2 Softmax Evolution

When softmax selection is used in TD methods, an action's probability of selection is a function of its estimated value. In addition to ensuring that the greedy action is chosen most often, this technique focuses exploration on the most promising alternatives. There are many ways to implement softmax selection but one popular method relies on a Boltzmann distribution [25], in which case an agent in state s chooses an action a with probability

$$\frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \quad (1)$$

where A is the set of available actions, $Q(s,a)$ is the agent's value estimate for the given state-action pair and τ is a positive parameter controlling the degree to which actions with higher values are favored in selection. The higher the value of τ, the more equiprobable the actions are.

As with ϵ-greedy selection, we use softmax selection in evolution to select policies for evaluation. At the beginning of each generation, each individual is evaluated for one episode, to initialize its fitness. Then, the remaining $e - |P|$ episodes are allocated according to a Boltzmann distribution. Before each episode, a policy p in a population P is selected with probability

$$\frac{e^{f(p)/\tau}}{\sum_{p' \in P} e^{f(p')/\tau}} \quad (2)$$

where $f(p)$ is the fitness of policy p, averaged over all the episodes for which it has been previously evaluated. In NEAT, softmax selection is applied in the same way as ϵ-greedy selection, except that the policy selected for evaluation is that returned by the softmax selection function described in Algorithm 3, where $e(p)$ is the total number of episodes for which a policy p has been evaluated so far.

Algorithm 3 SOFTMAX SELECTION(P, τ)

```
// P: population
// τ: softmax temperature

if ∃ p ∈ P | e(p) = 0 then
    return p
else
    total ← Σ_{p∈P} e^{f(p)/τ}
    for all p ∈ P do
        with-prob(e^{f(p)/τ} / total) return p
        else total ← total − e^{f(p)/τ}
```

Softmax selection provides a more nuanced balance between exploration and exploitation than ϵ-greedy because it focuses its exploration on the most promising alternative to the current best policy. Softmax selection can quickly

abandon poorly performing policies and prevent them from reducing the reward accrued during learning.

4.3 Interval Estimation Evolution

An important disadvantage of both ϵ-greedy and softmax selection is that they do not consider the uncertainty of the estimates on which they base their selections. One approach that addresses this shortcoming is interval estimation [13]. When used in TD methods, interval estimation computes a $(100-\alpha)\%$ confidence interval for the value of each available action. The agent always takes the action with the highest upper bound on this interval. Hence, this strategy favors actions with high estimated value and also focuses exploration on the most promising but uncertain actions. The α parameter controls the balance between exploration and exploitation, with smaller values generating greater exploration.

The same strategy can be employed within evolution to select policies for evaluation. At the beginning of each generation, each individual is evaluated for one episode, to initialize its fitness. Then, the remaining $e - |P|$ episodes are allocated to the policy that currently has the highest upper bound on its confidence interval. In NEAT, interval estimation is applied just as in ϵ-greedy and softmax selection, except that the policy selected for evaluation is that returned by the interval estimation function described in Algorithm 4, where $[0, z(x)]$ is an interval within which the area under the standard normal curve is x. $f(p)$, $\sigma(p)$ and $e(p)$ are the fitness, standard deviation, and number of episodes, respectively, for policy p.

Algorithm 4 INTERVAL ESTIMATION(P, α)

1: *// P: population, α: uncertainty in confidence interval*
2:
3: **if** $\exists\, p \in P \mid e(p) = 0$ **then**
4: return p
5: **else**
6: return $\text{argmax}_{p \in P}[f(p) + z(\frac{100-\alpha}{200})\frac{\sigma(p)}{\sqrt{e(p)}}]$

4.4 Results

As a baseline of comparison, I applied the original, off-line version of NEAT to both the mountain car and server job scheduling domains and averaged its performance over 25 runs. Next, I applied the ϵ-greedy, softmax, and interval estimation versions of NEAT to both domains using the same parameter settings.

Figure 3 summarizes the results of these experiments by plotting a uniform moving average over the last 1,000 episodes of the total reward accrued per episode for each method. I plot average reward because it is an on-line metric: it measures the amount of reward the agent accrues while it is learning. The best policies discovered by evolution, i.e. the generation champions, perform substantially higher than this average. However, using their performance as an evaluation metric would ignore the on-line cost that was incurred by evaluating the rest of population and receiving less reward per episode. Error bars on the graph indicate 95% confidence intervals. In addition, Student's t-tests confirm, with 95% confidence, the statistical significance of the performance difference between each pair of methods except softmax and interval estimation.

The results clearly demonstrate that selection mechanisms borrowed from TD methods can dramatically improve the on-line performance of evolutionary computation. All three on-line methods substantially outperform the off-line version of NEAT. In addition, the more nuanced strategies of softmax and interval estimation fare better than ϵ-greedy. This result is not surprising since the ϵ-greedy approach simply interleaves the search for better policies with exploitative episodes that employ the best known policy. Softmax selection and interval estimation, by contrast, concentrate exploration on the most promising alternatives. Hence, they spend fewer episodes on the weakest individuals and achieve better performance as a result.

The on-line methods, especially interval estimation, show a series of 10,000-episode intervals. Each of these intervals corresponds to one generation. The performance improvements within each generation reflect the on-line methods' ability to exploit the information gleaned from earlier episodes. As the generation progresses, these methods become better informed about which individuals to favor when exploiting and average reward increases as a result.

While these intervals reveal an important feature of the on-line methods' behavior, they can make it difficult to compare performance. For example, in the mountain car domain, interval estimation begins each generation with a lot of exploration and, consequently, relatively poor performance. However, that exploration quickly pays off and its average performance rises slightly above that of softmax. Which of these two methods is receiving more reward overall? It is difficult to tell from plots of average reward. Hence, Figure 4 plots, for the same experiments, the total cumulative reward accrued by each method over the entire run. As with the previous graph, error bars indicate 95% confidence intervals and Student's t-tests confirmed, with 95% confidence, the statistical significance of the performance difference between each pair of methods except softmax and interval estimation. Not surprisingly, the off-line version of NEAT accumulates much less reward than the on-line methods and ϵ-greedy accumulates less reward than the other on-line approaches. These graphs also show that, in mountain car, interval estimation's exploration early in each generation pays off, as it earns at least as much reward overall as softmax.

Overall, these results verify the efficacy of these methods of on-line evolution. It is less clear, however, which strategy is most useful. Softmax clearly outperforms ϵ-greedy but may be more difficult to use in practice because the τ parameter is harder to tune, as evidenced by the need to assign it different values in the two domains. As Sutton and Barto write, "Most people find it easier to set the ϵ parameter with confidence; setting τ requires knowledge of the likely action values and of powers of e." [25, pages 27-30]. In this light, interval estimation may be the best choice. Our experiments show that it performs as well or better than softmax and anecdotal evidence suggests that the α parameter is not overly troublesome to tune.

5. FUTURE WORK

There are many ways that the work presented in this thesis could be extended, refined, or further evaluated. This section enumerates a few of the possibilities.

5.1 Using Different Policy Search Methods

This thesis focuses on using evolutionary methods to automate the search for good function approximator representations. However, many other forms of policy search such as

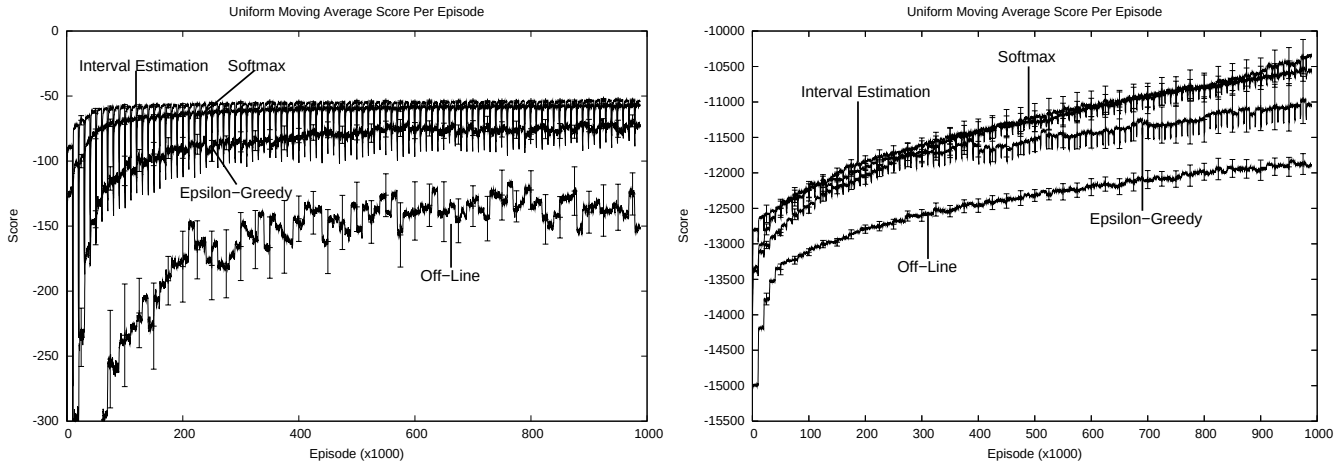


Figure 3: The uniform moving average reward accrued by off-line NEAT, compared to three versions of on-line NEAT in the mountain car and server job scheduling domains. In both domains, all rewards are negative so the agents strive to get average reward as close to zero as possible.

PEGASUS [17] and policy gradient methods [24, 14] have also succeeded on difficult reinforcement learning tasks. TD methods could be combined with these methods in the same way they are combined with evolutionary computation in this thesis. In the future, I plan to test some of these alternative combinations.

5.2 Reducing Sample Complexity

One disadvantage of evolutionary function approximation is its high sample complexity, since each fitness evaluation lasts for many episodes. However, in domains where the fitness function is not too noisy, each fitness evaluation could be conducted in a single episode if the candidate function approximator was pre-trained using methods based on experience replay [16]. By saving sample transitions from the previous generation, each new generation could be be trained off-line. This method would use much more computation time but many fewer sample episodes. Since sample experience is typically a much scarcer resource than computation time, this enhancement could greatly improve the practical applicability of evolutionary function approximation.

5.3 Addressing Non-Stationarity

In *non-stationary* domains, the environment can change in ways that alter the optimal policy. Since this phenomenon occurs in many real-world scenarios, it is important to develop methods that can handle it robustly. Evolutionary and TD methods are both well suited to non-stationary tasks and I expect them to retain that capability when combined. In fact, I hypothesize that evolutionary function approximation will adapt to non-stationary environments *better* than manual alternatives. If the environment changes in ways that alter the optimal representation, evolutionary function approximation can adapt, since it is continually testing different representations and retaining the best ones. By contrast, even if they are effective at the original task, manually designed representations cannot adapt in the face of changing environments.

On-line evolutionary computation should also excel in non-stationary environments, though some refinement will be necessary. The methods presented in this thesis implicitly assume a stationary environment because they compute the fitness of each individual by averaging over *all* episodes of evaluation. In non-stationary environments, older evaluations can become stale and misleading. Hence, fitness estimates should place less trust in older evaluations. This effect could easily be achieved using recency-weighting update rules like those employed by table-based TD methods.

5.4 Using Steady-State Evolutionary Computation

The NEAT algorithm used in this thesis is an example of *generational* evolutionary computation, in which an entire population is is evaluated before any new individuals are bred. Evolutionary function approximation might be improved by using a *steady-state* implementation instead [7]. Steady-state systems never replace an entire population at once. Instead, the population changes incrementally after each fitness evaluation, when one of the worst individuals is removed and replaced by a new offspring whose parents are among the best. Hence, an individual that receives a high score can more rapidly effect the search, since it immediately becomes a potential parent. In a generational system, that individual cannot breed until the beginning of the following generation, which might be thousands of episodes later. Hence, steady-state systems could help evolutionary function approximation perform better in on-line and non-stationary environments by speeding the adoption of new improvements. Fortunately, a steady-state version of NEAT already exists [20] so this extension is quite feasible.

6. REFERENCES

[1] D. Ackley and M. Littman. Interactions between learning and evolution. *Artificial Life II, SFI Studies in the Sciences of Complexity*, 10:487–509, 1991.

[2] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

[3] L. Baird and A. Moore. Gradient descent for general

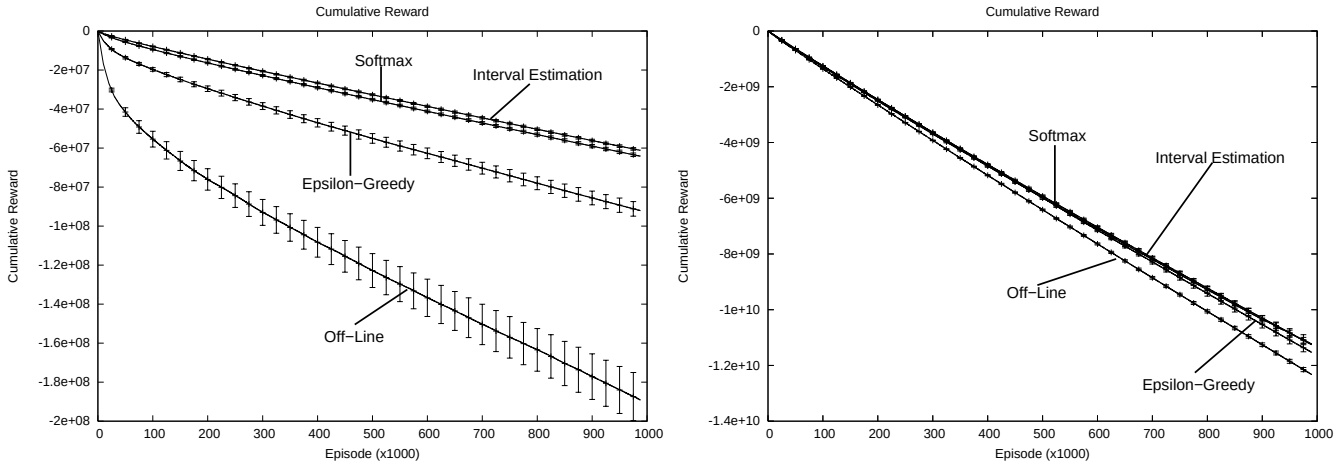


Figure 4: The cumulative reward accrued by off-line NEAT, compared to three versions of on-line NEAT in the mountain car and server job scheduling domains. In both domains, all rewards are negative so the agents strive to keep cumulative reward as close to zero as possible.

reinforcement learning. In *Advances in Neural Information Processing Systems 11*. MIT Press, 1999.

[4] J. M. Baldwin. A new factor in evolution. *The American Naturalist*, 30:441–451, 1896.

[5] E. Boers, M. Borst, and I. Sprinkhuizen-Kuyper. Evolving Artificial Neural Networks using the "Baldwin Effect". Technical Report TR 95-14, May 1995.

[6] R. H. Crites and A. G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2-3):235–262, 1998.

[7] T. C. Fogarty. An incremental genetic algorithm for real-time learning. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 416–419, 1989.

[8] R. French and A. Messinger. Genes, phenes and the Baldwin effect: Learning and evolution in a simulated population. *Artificial Life*, 4:277–282, 1994.

[9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1989.

[10] F. Gruau and D. Whitley. Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. *Evolutionary Computation*, 1:213–233, 1993.

[11] 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.

[12] G. E. Hinton and S. J. Nowlan. How learning can guide evolution. *Complex Systems*, 1:495–502, 1987.

[13] L. P. Kaelbling. *Learning in Embedded System*. MIT Press, 1993.

[14] N. Kohl and P. Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

[15] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4(2003):1107–1149, 2003.

[16] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[17] A. Y. Ng and M. I. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 406–415. Morgan Kaufmann Publishers Inc., 2000.

[18] S. Nolfi, J. L. Elman, and D. Parisi. Learning and evolution in neural networks. *Adaptive Behavior*, 2:5–28, 1994.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, pages 318–362. 1986.

[20] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. Evolving neural network agents in the NERO video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, 2005.

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

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

[23] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[24] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. A. Solla, T. K. Leen, and K.-R. Muller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 1057–1063. The MIT Press, 2000.

[25] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[26] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[27] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.

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