In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO 06), pp. 1577-1584, Seattle, WA, July 2006.

On-Line Evolutionary Computation for Reinforcement Learning in Stochastic Domains

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

Peter Stone
Department of Computer Sciences
University of Texas at Austin
1 University Station, C0500
Austin, TX 78712-0233
pstone@cs.utexas.edu

ABSTRACT

In *reinforcement learning*, an agent interacting with its environment strives to learn a policy that specifies, for each state it may encounter, what action to take. Evolutionary computation is one of the most promising approaches to reinforcement learning but its success is largely restricted to *off-line* scenarios. In *on-line* scenarios, an agent must strive to maximize the reward it accrues *while it is learning*. *Temporal difference* (TD) methods, another approach to reinforcement learning, naturally excel in on-line scenarios because they have selection mechanisms for balancing the need to search for better policies (*exploration*) with the need to accrue maximal reward (*exploitation*). This paper presents a novel way to strike this balance in evolutionary methods by borrowing the selection mechanisms used by TD methods to choose individual actions and using them in evolution to choose policies for evaluation. Empirical results in the mountain car and server job scheduling domains demonstrate that these techniques can substantially improve evolution's on-line performance in stochastic domains.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

evolutionary computation, reinforcement learning, neural networks, on-line learning

1. INTRODUCTION

In *reinforcement learning*, an agent interacting with its environment strives to learn a policy that specifies, for each state it may encounter, what action to take. Unlike supervised learning tasks, the agent never sees examples of correct behavior. Instead, it receives only positive or negative rewards for the actions it tries. From this feedback it must find a policy that maximizes reward over the long-term.

Evolutionary computation is one of the most promising approaches to tackling reinforcement learning problems. Evolutionary methods have succeeded in a wide range of domains, including benchmark pole balancing tasks [19], game playing [21], and robot control [20]. However, this empirical success is largely restricted to *off-line* scenarios, in which the agent learns, not in the real-world, but in a "safe" environment like a simulator. Consequently, its only goal is to learn a good policy as quickly as possible. It does not care how much reward it accrues *while it is learning* because those rewards are only hypothetical and do not correspond to real-world costs. If the agent tries disastrous policies, only computation time is lost.

Unfortunately, many reinforcement learning problems cannot be solved off-line because no simulator is available. Sometimes the dynamics of the task are unknown, e.g. when a robot explores an unfamiliar environment or a chess player plays a new opponent. Other times, the dynamics of the task are too complex to accurately simulate, e.g. user behavior on a large computer network or the noise in a robot's sensors and actuators. In such domains, the agent has no choice but to learn *on-line*: by interacting with the real world and adjusting its policy as it goes. In an on-line learning scenario, it is not enough for an agent to learn a good policy quickly. It must also maximize the reward it accrues while it is learning because those rewards correspond to real-world costs. For example, if a robot learning on-line tries a policy that causes it to drive off a cliff, then the negative reward the agent receives is not hypothetical; it corresponds to the very real cost of fixing or replacing the robot.[1]

To excel in on-line scenarios, a learning algorithm must effectively balance two competing objectives. The first objective is *exploration*, in which the agent tries alternatives to its current best policy in the hopes of improving it. The second objective is *exploitation*, in which the agent follows the current best policy in order to maximize the reward it receives. Evolutionary methods already strive to perform this balance. In fact, Holland [8] argues that the reproduction mechanism encourages exploration, since crossover and mutation result in novel genomes, but also encourages exploitation, since each new generation is based on the fittest members of the last one. However, reproduction allows evolutionary methods to balance exploration and exploitation only *across* generations, not *within* them. Once the mem-

[1]The term *on-line learning* is sometimes used in a very different way: to refer to *non-stationary* learning problems where the agent's environment is changing in ways that alter the optimal policy. In such problems, the agent must continually adapt to perform well. The problems of non-stationary learning and on-line learning (as we use the term) are unrelated and orthogonal: a learning scenario can be non-stationary but off-line or stationary but on-line. This paper does not address non-stationary learning problems.

bers of each generation have been determined, they all typically receive the same evaluation time.

This approach makes sense in deterministic domains, where each member of the population can be accurately evaluated in a single episode. However, most real-world domains are stochastic, in which case fitness evaluations must be averaged over many episodes. In these domains, giving the same evaluation time to each member of the population can be grossly suboptimal because, within a generation, it is purely exploratory. Instead, an on-line evolutionary algorithm should exploit the information gained earlier in the generation to systematically give more evaluations to the most promising individuals and avoid re-evaluating the weakest ones. Doing so allows evolutionary methods to increase the reward accrued during learning.

This paper presents a novel approach to achieving this balance that relies on action selection mechanisms typically used by *temporal difference* (TD) methods [24]. TD methods, which solve reinforcement learning problems by estimating the agent's optimal value function, naturally excel in on-line scenarios because they use action selection mechanisms to control how often the agent exploits (by behaving greedily with respect to current value estimates) and how often it explores (by trying alternative actions). This paper describes ways to borrow the selection mechanisms used by TD methods to choose individual actions and use them in evolution to choose policies for evaluation. This approach enables evolution to excel on-line by balancing exploration and exploitation within *and* across generations.

This paper investigates three methods based on this approach. The first, based on ϵ-greedy selection [26], switches probabilistically between searching for better policies and re-evaluating the best known policy. The second, based on softmax selection [24], distributes evaluations in proportion to each individual's estimated fitness. The third, based on interval estimation [9], computes confidence intervals for the fitness of each policy and always evaluates the policy with the highest upper bound.

To evaluate these methods, we implemented them in NEAT, a neuroevolutionary method known to excel at reinforcement learning tasks [19], and tested their performance in two domains: 1) mountain car, a canonical reinforcement learning benchmark task, and 2) server job scheduling, a large stochastic reinforcement learning task from the field of *autonomic computing* [10]. The results demonstrate that these techniques can substantially improve the on-line performance of evolutionary methods and that softmax selection and interval estimation are more effective than the simple ϵ-greedy approach.

2. NEUROEVOLUTION OF AUGMENTING TOPOLOGIES (NEAT)[2]

The experiments presented in this paper use NeuroEvolution of Augmenting Topologies (NEAT) as a representative evolutionary method for testing different approaches to on-line evolution. NEAT is an appropriate choice because of its empirical successes on reinforcement learning tasks like pole balancing [19], game playing [21], and robot control [20].

In a typical neuroevolutionary system [29], 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 the NEAT method. Stanley and Miikkulainen [19] present a full description.

2.1 Minimizing Dimensionality

Unlike other systems that evolve network topologies and weights [7, 29], 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.

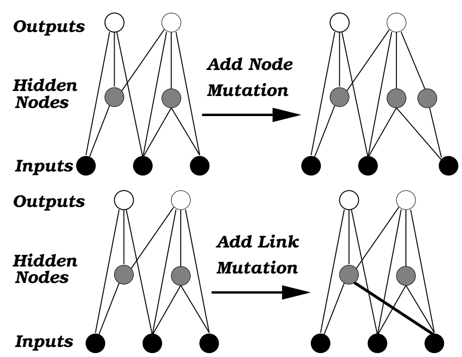


Figure 1: Examples of NEAT's mutation operators for adding structure to networks. At top, a new hidden node splits a link in two. At bottom, a new link, shown with a thicker black line, connects two nodes.

2.2 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.

In order 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. When 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. When these genomes crossover, innovation numbers on inherited genes are preserved. Thus, the historical origin of every gene in the system is known throughout evolution.

[2] This section is adapted from the original NEAT paper [19].

Through innovation numbers, the system knows exactly 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. During crossover, 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.

2.3 Speciation

In most cases, adding new structure to a network initially reduces its fitness. However, NEAT speciates the population, 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* [6], where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

2.4 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 solve reinforcement learning tasks. 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 during an episode and observing how much reward it receives. The policy's fitness is simply the sum of the rewards the agent accrues during the episode. In deterministic domains, each member of the population can be evaluated in a single episode. However, most real-world problems are stochastic and the reward a particular policy receives on a given episode has substantial variance. To reduce this variance, evolution must evaluate each candidate policy many times and average the resulting fitness estimates. In such domains, the question arises: how should episodes of evaluation be allocated among the population so as to maximize on-line performance? The following section presents a novel answer.

3. 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, we 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. However, the 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 evolution 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.

3.1 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.

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 1. This function returns a policy p which is either selected randomly or which maximizes $f(p)$, the fitness of p averaged over all the episodes for which it has been previously evaluated.

Algorithm 1 ϵ-GREEDY SELECTION(P, ϵ)

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

with-prob(ϵ) return RANDOM(P)
else return $\text{argmax}_{p \in P} f(p)$

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.

3.2 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 [24], in which case an agent in state s chooses an action a with probability

$$\frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \tag{2}$$

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

$$\frac{e^{f(p)/\tau}}{\sum_{p' \in P} e^{f(p')/\tau}} \tag{3}$$

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 2, where $e(p)$ is the total number of episodes for which a policy p has been evaluated so far.

Algorithm 2 SOFTMAX SELECTION(P, τ)

// P: population, τ: softmax temperature

if $\exists\, p \in P \mid e(p) = 0$ **then**
  return p
else
  $total \leftarrow \sum_{p \in P} e^{f(p)/\tau}$
  for all $p \in P$ **do**
    with-prob$(\frac{e^{f(p)/\tau}}{total})$ return p
    else $total \leftarrow total - e^{f(p)/\tau}$

Softmax selection provides a more nuanced balance between exploration and exploitation than ϵ-greedy because it focuses its exploration on the most promising alternatives to the current best policy. Softmax selection can quickly abandon poorly performing policies and prevent them from reducing the reward accrued during learning.

3.3 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 [9]. 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. 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 3, 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 3 INTERVAL ESTIMATION(P, α)

// P: population, α: uncertainty in confidence interval

if $\exists\, p \in P \mid e(p) = 0$ **then**
  return p
else
  return $\text{argmax}_{p \in P}[f(p) + z(\frac{100-\alpha}{200})\frac{\sigma(p)}{\sqrt{e(p)}}]$

4. EXPERIMENTAL SETUP

To empirically compare the methods described above, we used two different reinforcement learning domains. The first domain, mountain car, is a standard reinforcement learning benchmark task. The second domain, server job scheduling, is a large, stochastic domain from the field of autonomic computing.

4.1 Mountain Car

In the mountain car task [4], depicted in Figure 2, an agent strives to drive a car to the top of a steep mountain. The car cannot simply accelerate forward because its engine is not powerful enough to overcome gravity. Instead, the agent must learn to drive backwards up the hill behind it, thus building up sufficient inertia to ascend to the goal before running out of speed.

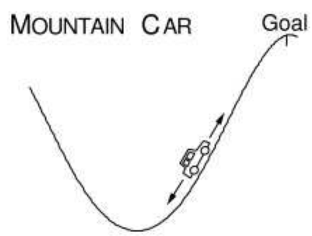


Figure 2: The mountain car task. This figure was taken from Sutton and Barto [21].

The agent's state at timestep t consists of its current position p_t and its current velocity v_t. It receives a reward of -1 at each time step until reaching the goal, at which point the episode terminates. The agent's three available actions correspond to the throttle settings 1, 0, and -1. The following equations control the car's movement:

$$p_{t+1} = bound_p(p_t + v_{t+1})$$

$$v_{t+1} = bound_v(v_t + 0.001a_t - 0.0025cos(3p_t))$$

where a_t is the action the agent takes at timestep t, $bound_p$ enforces $-1.2 \leq p_{t+1} \leq 0.5$, and $bound_v$ enforces $-0.07 \leq v_{t+1} \leq 0.07$. In each episode, the agent begins in a state chosen randomly from these ranges. To prevent episodes from running indefinitely, each episode is terminated after 2,500 steps if the agent still has not reached the goal.

To represent the agent's current state to the network, we divided each state feature into ten regions. One input was associated with each region (for a total of twenty inputs) and was set to one if the agent's current state fell in that region and to zero otherwise. Hence, only two inputs were activated for any given state. The networks have three outputs, each corresponding to one of the actions available to the agent.

4.2 Server Job Scheduling

While the mountain car task is a useful benchmark, it is a very simple domain. To assess whether on-line evolution performs well in a much more complex problem, we use a challenging reinforcement learning task called server job scheduling. This domain is drawn from the burgeoning field of autonomic computing [10]. The goal of autonomic computing is to develop computer systems that automatically configure themselves, optimize their own behavior, and diagnose and repair their own failures.

One important autonomic computing task is server job scheduling [27], in which a server, such as a website's application server or database, must determine in what order to process the jobs currently waiting in its queue. Its goal is to maximize the aggregate utility of all the jobs it processes. A *utility function* for each job type maps the job's completion time to the utility derived by the user [25]. The problem of server job scheduling becomes challenging when these utility functions are non-linear and/or the server must process multiple types of jobs. Since selecting a particular job for processing necessarily delays the completion of all other jobs in the queue, the scheduler must weigh difficult trade-offs to maximize aggregate utility.

Our experiments were conducted in a Java-based simulator. During each timestep, the server removes one job from its queue and completes it. During each of the first 100 timesteps, a new job of a randomly selected type is added to the end of the queue. Hence, the agent must make decisions about which job to process next even as new jobs are arriving. The simulation begins with 100 jobs preloaded into the server's queue and ends when the queue becomes empty. Since one job is processed at each timestep, each episode lasts 200 timesteps. For each job that completes, the scheduling agent receives an immediate reward determined by that job's utility function.

Utility functions for the four job types used in our experiments are shown in Figure 3. Users who create jobs of type #1 or #2 do not care about their jobs' completion times so long as they are less than 100 timesteps. Beyond that, they get increasingly unhappy. The rate of this change differs between the two types and switches at timestep 150. Users who create jobs of type #3 or #4 want their jobs completed as quickly as possible. However, once the job becomes 100 timesteps old, it is too late to be useful and they become indifferent to it. As with the first two job types, the slopes for job types #3 and #4 differ from each other and switch, this time at timestep 50. Note that all these utilities are negative functions of completion time. Hence, the scheduling agent strives to bring aggregate utility as close to zero as possible.

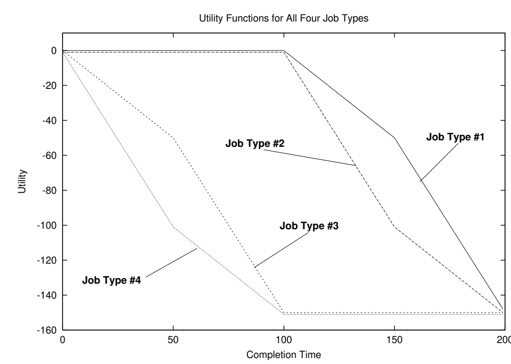
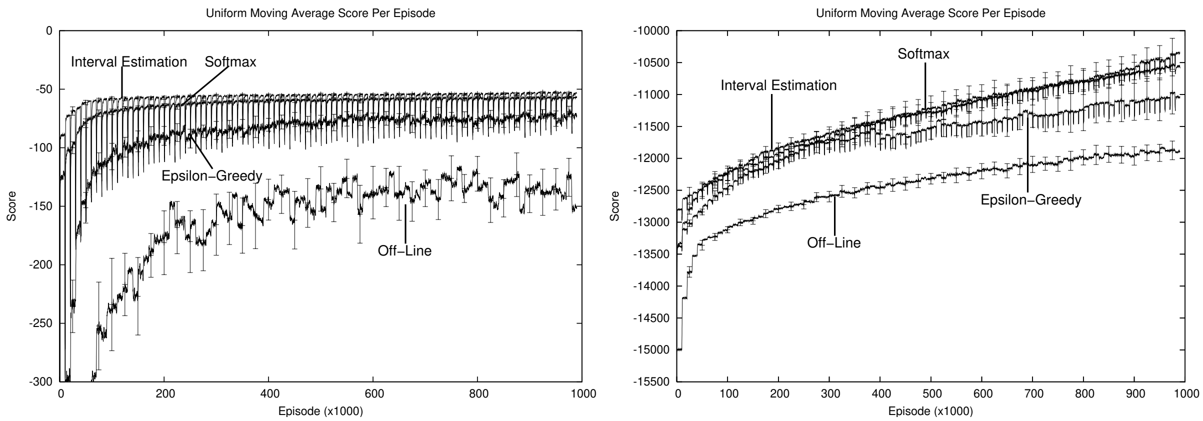


Figure 3: The four utility functions.

To make the state and action spaces more manageable, we discretize them. The range of job ages from 0 to 200 is divided into four sections and the scheduler is told, at each timestep, how many jobs in the queue of each type fall in each range, resulting in 16 state features. The action space is similarly discretized. Instead of selecting a particular job for processing, the scheduler specifies what type of job it wants to process and which of the four age ranges that job should lie in, resulting in 16 distinct actions.

The server job scheduling domain is a perfect example of a reinforcement learning task that needs to be solved on-line. Though we use a simulator for the purpose of experimental research, creating an accurate simulator in the real world would not be practical. Such a simulator would have to precisely model the server's internal workings and the behavior of all the system's users, including how that behavior changes in response to different scheduling policies. Hence,



(a) Mountain Car (b) Server Job Scheduling

Figure 4: 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.

good policies can probably only be learned on-line, by trying them out on real servers. In such scenarios, maximizing on-line performance is critical, since lost reward corresponds to delays for real users.

5. RESULTS AND DISCUSSION

As a baseline of comparison, we 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. The population size $|P|$ was 100 and the number of episodes per generation e was 10,000. Hence, each member of the population was evaluated for 100 episodes. See Appendix A for more details on the NEAT parameters used in our experiments. Next, we applied the ϵ-greedy, softmax, and interval estimation versions of NEAT to both domains using the same parameter settings. Each of these on-line methods has associated with it one additional parameter which controls the balance between exploration and exploitation. For each method, we experimented informally with approximately ten different settings of these parameters to find ones that worked well in the two tasks. Finally, we averaged the performance of each method over 25 runs using the best known parameter settings.

Those settings were as follows. For ϵ-greedy, ϵ was set to 0.25. This value is larger than is typically used in TD methods but makes intuitive sense, since exploration in NEAT is safer than in TD methods. After all, even when NEAT explores, the policies it selects are not drawn randomly from policy space. On the contrary, they are the children of the previous generation's fittest parents. For softmax, the appropriate value of τ depends on the range of fitness scores, which differs dramatically between the two domains. Hence, different values were required for the two domains: we set τ to 50 in mountain car and 500 in server job scheduling. For interval estimation, α was set to 20, resulting in 80% confidence intervals.

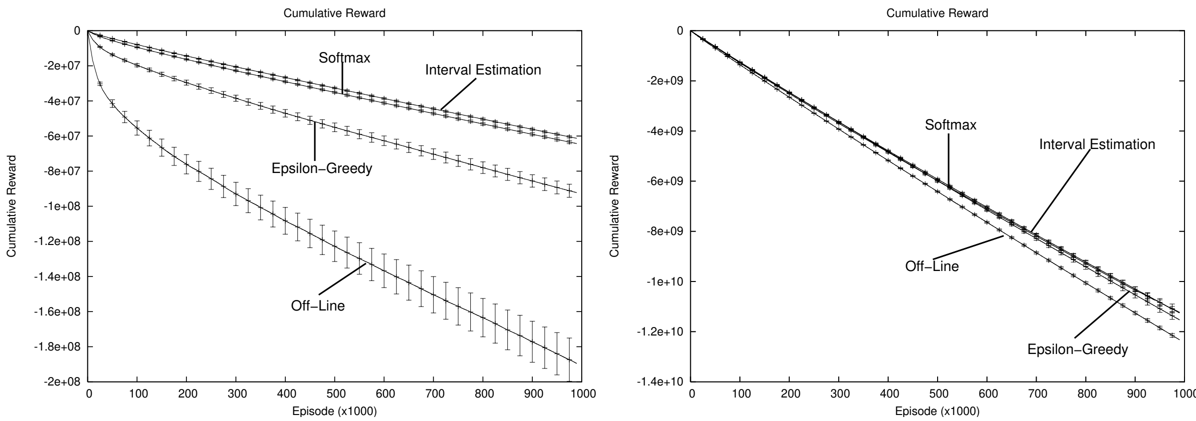
Figure 4 summarizes the results of these experiments by plotting a uniform moving average over the last 1,000 episodes of the total reward accrued per episode for each method. We plot average reward because it is an on-line metric: it measures the amount of reward the agent accrues while it is learning. 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.[3] 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 5 plots, for the same experiments, the total cumulative reward

[3]Though detailed comparisons with TD methods are beyond the scope of this paper, in other experiments we find that for these tasks both off-line and on-line NEAT outperform TD methods such as Q-learning and Sarsa with neural network function approximators [27].



(a) Mountain Car (b) Server Job Scheduling

Figure 5: 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.

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.

Together, these results demonstrate that borrowing selection mechanisms from TD methods can greatly improve the on-line performance of evolutionary computation. However, they do not address how on-line evolution affects the quality of the best policies discovered. Does excelling at on-line metrics necessarily hurt performance on off-line metrics? To answer this question, we selected the best policies discovered by each method (i.e. the final generation champions) and evaluated them each for 1,000 additional episodes.

In mountain car, using on-line evolution has no noticeable effect: the best policies of off-line and all three versions of on-line NEAT receive an average score of approximately -52, which matches the best results achieved in previous research on this domain [16, 23]. While the mountain car domain is simple enough that all the methods find approximately optimal policies, the same is not true in scheduling, where ϵ-greedy performs substantially worse. Its best policies receive an average score of approximately -11,100, whereas off-line and the other two versions of on-line NEAT all receive an average score of approximately -10,100. This result is not surprising: since ϵ-greedy evolution spends most of its episodes re-evaluating the best policy, its fitness estimates for the rest of the population are less accurate. By focusing exploration on the most promising individuals, softmax and interval estimation offer the best of both worlds: they excel at the on-line metrics without sacrificing the quality of the best policies discovered.

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." [24, 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.

6. RELATED AND FUTURE WORK

The approach presented in this paper is closely related to Learning Classifier Systems [12] in that they are often applied to reinforcement learning problems and make use of TD methodology. NCS [5] is particularly related because it uses neural networks, as are "Pittsburgh-style" classifiers [17] because, as in our approach, each member of the population represents a complete candidate solution. The primary difference between classifier systems and our approach is that the former strive to learn value functions whereas the latter learns policies directly. Selection mechanisms like ϵ-greedy have been applied to classifier systems to manage the exploration/exploitation trade-off (e.g. [11, 14, 28]). However, such mechanisms are typically used, as in TD methods, to select among individual actions, not to allocate evaluations among an entire population.

Because it allows members of the same population to receive different numbers of evaluations, this research is also similar to previous work about optimizing noisy fitness functions. For example, Stagge [18] introduces mechanisms for deciding which individuals need more evaluations, assuming the noise is Gaussian. Beielstein and Markon [2] use a similar approach to develop tests for determining which individuals should survive. However, this area of research has a significantly different focus, since the goal is to find the best individuals using the fewest evaluations, not to maximize the reward accrued during those evaluations.

The problem of using evolutionary systems on-line is more closely related to other research about the tradeoff between exploration and exploitation, which has been studied extensively in the context of TD methods [24, 26] and multiarmed bandit problems [1, 3, 13]. The selection mechanisms used in this paper are well-established though, to our knowledge, their application to evolutionary computation is novel.

In future work, we aim to apply the TD selection mechanisms used here to other policy search methods. Nothing in our methodology requires the underlying reinforcement learning method to be evolutionary. Hence, we hypothesize that other policy search techniques (e.g. policy gradient methods [22]) could also improve their on-line performance via the methods studied in this paper.

7. CONCLUSION

We present a novel way of boosting the on-line performance of evolutionary methods by borrowing selection mechanisms used in TD methods to choose individual actions and using them in evolution to choose policies for evaluation. Empirical results in the mountain car and server job scheduling domains demonstrate that these techniques can substantially improve the on-line performance of evolutionary methods and that softmax selection and interval estimation are more effective than the simple ϵ-greedy approach.

Acknowledgements

We would like to thank Ken Stanley, Nate Kohl, Risto Miikkulainen and the anonymous reviewers for their insightful comments about initial versions of this paper. This research was supported in part by NSF CAREER award IIS-0237699 and an IBM faculty award.

APPENDIX

A. NEAT PARAMETERS

This section details the NEAT parameters used in our experiments. Stanley and Miikkulainen [19] describes these parameters in detail. The coefficients for measuring compatibility were $c_1 = 1.0$, $c_2 = 1.0$, and $c_3 = 2.0$. The survival threshold was 0.2. The champion of each species with more than five networks was copied into the next generation unchanged. The weight mutation power was 0.005 in mountain car and 0.5 in scheduling. The interspecies mating rate was 0.01. The number of target species was 6. The drop-off age was 100. The probability of adding recurrent links was 0.0.

B. REFERENCES

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

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

[3] R. E. Bellman. A problem in the sequential design of experiments. *Sankhya*, 16:221–229, 1956.

[4] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, 1995.

[5] L. Bull and J. Hurst. A neural learning classifier system with self-adaptive constructivism. In *GECCO-03: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, 2003.

[6] 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.

[7] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, 1996.

[8] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. University of Michigan Press, Ann Arbor, MI, 1975.

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

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[11] P. L. Lanzi and M. Colombetti. An extension to the XCS classifier system for stochastic environments. In *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 353–360, 1999.

[12] P. L. Lanzi, W. Stolzmann, and S. Wilson. *Learning classifier systems from foundations to applications*. Springer, 2000.

[13] W. G. Macready and D. H. Wolpert. Bandit problems and the exploration/exploitation tradeoff. In *IEEE Transactions on Evolutionary Computation*, volume 2(1), pages 2–22, 1998.

[14] A. McMahon, D. Scott, and W. N. Browne. An autonomous explore/exploit strategy. In *GECCO-05: Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program*, 2005.

[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] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeeth International Conference on Machine Learning*, pages 903–910, 2000.

[17] S. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of the Eighth International Joint Conference on Articial Intelligence*, pages 422–425, 1983.

[18] P. Stagge. Averaging efficiently in the presence of noise. In *Parallel Problem Solving from Nature*, volume 5, pages 188–197, 1998.

[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] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation.

[23] 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.

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

[25] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing*, pages 70–77, 2004.

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

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

[28] S. W. Wilson. Explore/exploit strategies in autonomy. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, 1996.

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