To appear in *Journal of Machine Learning Research*, 2006.

Evolutionary Function Approximation for Reinforcement Learning

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

Editor: Sven Koenig, Shie Mannor, and Georgios Theocharous

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 paper 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*. We 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 paper 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. We 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. This paper also presents additional tests that offer insight into what factors can make neural network function approximation difficult in practice.

Keywords: reinforcement learning, temporal difference methods, evolutionary computation, neuroevolution, on-line learning

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) Sutton (1988), 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 Sutton and Barto (1998). 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 Baird (1995) and agents that perform poorly. Even for reinforcement learning algorithms with guaranteed convergence Baird and Moore (1999); Lagoudakis and Parr (2003), 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." Lagoudakis and Parr (2003, p. 1111) Nonetheless, representational choices are typically made manually, based only on the designer's intuition.

Our goal is to automate the search for effective representations by employing sophisticated optimization techniques. In this paper, we focus on using evolutionary methods Goldberg (1989) because of their demonstrated ability to discover effective representations Gruau et al. (1996); Stanley and Miikkulainen (2002). 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 Hinton and Nowlan (1987); Ackley and Littman (1991); Boers et al. (1995); French and Messinger (1994); Gruau and Whitley (1993); Nolfi et al. (1994) but never, to our knowledge, to aid the discovery of good TD function approximators.

Our approach requires only 1) an evolutionary algorithm capable of optimizing representations from a class of functions and 2) a TD method that uses elements of that class for function approximation. This paper focuses on performing evolutionary function approximation with neural networks. There are several reasons for this choice. First, they have great experimental value. Nonlinear function approximators are often the most challenging to use; hence, success for evolutionary function approximation with neural networks is good reason to hope for success with linear methods too. Second, neural networks have great potential, since they can represent value functions linear methods cannot (given the same basis functions). Finally, employing neural networks is feasible because they have previously succeeded as TD function approximators Crites and Barto (1998); Tesauro (1994) and sophisticated methods for optimizing their representations Gruau et al. (1996); Stanley and Miikkulainen (2002) already exist.

This paper uses NeuroEvolution of Augmenting Topologies (NEAT) Stanley and Miikkulainen (2002) to select neural network function approximators for Q-learning Watkins (1989), 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 and softmax selection Sutton and Barto (1998). 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 paper investigates a novel approach we call *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. We present two implementations, based on ϵ-greedy and softmax selection, that distribute evaluations within a generation so as to favor more promising individuals. Since on-line evolutionary computation can be used in conjunction with evolutionary function approximation, the ability to optimize representations need not come at the expense of the on-line aspects of TD methods. On the contrary, the value function and its representation can be optimized simultaneously, all while the agent interacts with its environment.

We evaluate these contributions with extended empirical studies in two domains: 1) mountain car and 2) server job scheduling. The mountain car task Sutton and Barto (1998) is a canonical reinforcement learning benchmark domain that requires function approximation. Though the task is simple, previous researchers have noted that manually designed neural network function approximators are often unable to master it Boyan and Moore (1995); Pyeatt and Howe (2001). Hence, this domain is ideal for a preliminary evaluation of NEAT+Q.

Server job scheduling Whiteson and Stone (2004), is a large, probabilistic reinforcement learning task from the field of *autonomic computing* Kephart and Chess (2003). In server job scheduling, a server, such as a website's application server or database, must determine in what order to process a queue of waiting jobs so as to maximize the system's aggregate utility. This domain is challenging because it is large (the size of both the state and action spaces grow in direct proportion to the size of the queue) and probabilistic (the server does not know what type of job will arrive next). Hence, it is a typical example of a reinforcement learning task that requires effective function approximation.

Using these domains, our 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, we test NEAT and NEAT+Q with and without ϵ-greedy and softmax versions of evolutionary computation. These experiments confirm that such techniques can significantly improve the on-line performance of evolutionary computation. Finally, we present additional tests that measure the effect of continual learning on function approximators. The results offer insight into why certain methods outperform others in these domains and what factors can make neural network function approximation difficult in practice.

We view the impact of this work as two-fold. First, it provides a much-needed practical approach to selecting TD function approximators, automating a critical design step that is typically performed manually. Second, it provides an objective analysis of the strengths and weaknesses of evolutionary and TD methods, opportunistically combining the strengths into a single approach. Though the TD and evolutionary communities are mostly disjoint and focus on somewhat different problems, we find that each can benefit from the progress of the other. On the one hand, we show that methods for evolving neural network topologies can find TD function approximators that perform better. On the other hand, we show that established techniques from the TD community can make evolutionary methods applicable to on-line learning problems.

The remainder of this paper is organized as follows. Section 2 provides background on Q-learning and NEAT, the constituent learning methods used in this paper. Section 3 introduces the novel methods and details the particular implementations we tested. Section 4 describes the mountain car and server job scheduling domains and Section 5 presents and discusses empirical results. Section 7 overviews related work, Section 8 outlines opportunities for future work, and Section 9 concludes.

2. Background

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

2.1 Q-Learning

There are several different TD methods currently in use, including Q-learning Watkins (1989), Sarsa Sutton and Barto (1998), and LSPI Lagoudakis and Parr (2003). The experiments presented in this paper use Q-learning because it is a well-established, canonical method that has also enjoyed empirical success, particularly when combined with neural network function approximators Crites and Barto (1998). We present it as a representative method but do not claim it is superior to other TD approaches. In principle, evolutionary function approximation can be used with any of them. For example, many of the experiments described in Section 5 have been replicated with Sarsa Sutton and Barto (1998), another popular TD method, in place of Q-learning, yielding qualitatively similar results.

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 is defined by 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.

Algorithm 1 describes the Q-learning algorithm when a neural network is used to approximate the value function. 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 (line 5). The EVAL-NET function (line 9) returns the activation on the network's outputs after the given inputs are fed to the network and propagated forward. Since the network uses a sigmoid activation function, these values will all be in $[0, 1]$ and hence are rescaled according to a parameter k. At each step, the weights of the neural network are adjusted (line 13) such that its output better matches the current value estimate for the state-action pair: $r + \gamma \max_{a'} Q(s', a')$. The adjustments are made via the BACKPROP function, which implements the standard backpropagation algorithm Rumelhart et al. (1986) with the addition of accumulating eligibility traces controlled by λ Sutton and Barto (1998). The agent uses ϵ-greedy selection Sutton and Barto (1998) to ensure it occasionally tests alternatives to its current policy (lines 10–11). The agent interacts with the environment via the TAKE-ACTION function (line 15), which returns a reward and a new state.

Algorithm 1 Q-LEARN$(S, A, \sigma, c, \alpha, \gamma, \lambda, \epsilon_{td}, e)$

1: *// S: set of all states, A: set of all actions, σ: standard deviation of initial weights*
2: *// c: output scale, α: learning rate, γ: discount factor, λ: eligibility decay rate*
3: *// ϵ_{td}: exploration rate, e: total number of episodes*
4:
5: $N \leftarrow$ INIT-NET(S, A, σ) *// make a new network N with random weights*
6: **for** $i \leftarrow 1$ to e **do**
7:     $s, s' \leftarrow$ null, INIT-STATE(S) *// environment picks episode's initial state*
8:     **repeat**
9:         $Q[] \leftarrow c \times$EVAL-NET(N, s') *// compute value estimates for current state*
10:         **with-prob**(ϵ_{td}) $a' \leftarrow$ RANDOM(A) *// select random exploratory action*
11:         **else** $a' \leftarrow \text{argmax}_j Q[j]$ *// or select greedy action*
12:         **if** $s \neq$ null **then**
13:             BACKPROP$(N, s, a, (r + \gamma \max_j Q[j])/c, \alpha, \gamma, \lambda)$ *// adjust weights toward target*
14:         $s, a \leftarrow s', a'$
15:         $r, s' \leftarrow$ TAKE-ACTION(a') *// take action and transition to new state*
16:     **until** TERMINAL-STATE?(s)

2.2 NEAT[1]

The implementation of evolutionary function approximation presented in this paper 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 non-Markovian double pole balancing Stanley and Miikkulainen (2002), game

1. This section is adapted from the original NEAT paper Stanley and Miikkulainen (2002).

playing Stanley and Miikkulainen (2004b), and robot control Stanley and Miikkulainen (2004a), and because of its ability to automatically optimize network topologies.

In a typical neuroevolutionary system Yao (1999), 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.

NEAT is an optimization technique that can be applied to a wide variety of problems. Section 3 below describes how we use NEAT to optimize the topology and initial weights of TD function approximators. Here, we describe how NEAT can be used to tackle reinforcement learning problems without the aid of TD methods, an approach that serves as one baseline of comparison in Section 5. For this method, NEAT does not attempt to learn a value function. Instead, it finds good policies directly by training *action selectors*, which map states to the action the agent should take in that state. Hence it is an example of *policy search* reinforcement learning. Like other policy search methods, e.g. Sutton et al. (2000); Ng and Jordan (2000); Mannor et al. (2003); Kohl and Stone (2004), it uses global optimization techniques to directly search the space of potential policies.

Algorithm 2 NEAT$(S, A, p, m_n, m_l, g, e)$

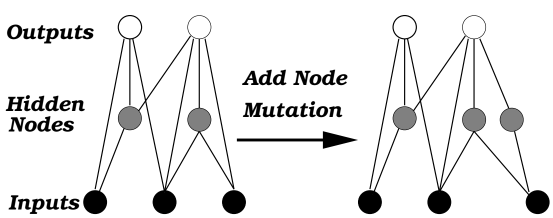
// *S: set of all states, A: set of all actions, p: population size, m_n: node mutation rate*
// *m_l: link mutation rate, g: number of generations, e: episodes per generation*

$P[] \leftarrow$ INIT-POPULATION(S, A, p) // *create new population P with random networks*
for $i \leftarrow 1$ to g **do**
  for $j \leftarrow 1$ to e **do**
    $N, s, s' \leftarrow$ RANDOM$(P[])$, null, INIT-STATE(S) // *select a network randomly*
    repeat
      $Q[] \leftarrow$ EVAL-NET(N, s') // *evaluate selected network on current state*
      $a' \leftarrow \mathrm{argmax}_i Q[i]$ // *select action with highest activation*
      $s, a \leftarrow s', a'$
      $r, s' \leftarrow$ TAKE-ACTION(a') // *take action and transition to new state*
      $N.fitness \leftarrow N.fitness + r$ // *update total reward accrued by N*
    until TERMINAL-STATE?(s)
    $N.episodes \leftarrow N.episodes + 1$ // *update total number of episodes for N*
  $P'[] \leftarrow$ new array of size p // *new array will store next generation*
  for $j \leftarrow 1$ to p **do**
    $P'[j] \leftarrow$ BREED-NET$(P[])$ // *make a new network based on fit parents in P*
    with-probability m_n: ADD-NODE-MUTATION$(P'[j])$ // *add a node to new network*
    with-probability m_l: ADD-LINK-MUTATION$(P'[j])$ // *add a link to new network*
  $P[] \leftarrow P'[]$

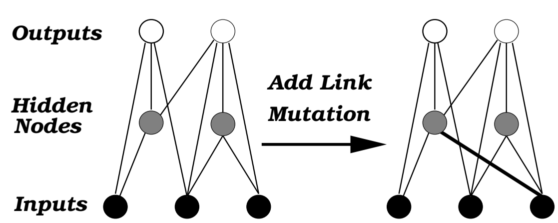
Algorithm 2 contains a high-level description of the NEAT algorithm applied to an episodic reinforcement learning problem. This implementation differs slightly from previous versions of NEAT in that evaluations are conducted by randomly selecting individuals (line 7), instead of the more typical approach of stepping through the population in a fixed order. This change does not significantly alter NEAT's behavior but facilitates the alterations we introduce in Section 3.2. During each step, the agent takes whatever action corresponds to the output with the highest activation (lines 10–12). NEAT maintains a running total of the reward accrued by the network during its evaluation (line 13). Each generation ends after e episodes, at which point each network's average fitness is $N.fitness/N.episodes$. In stochastic domains, e typically must be much larger than $|P|$ to ensure accurate fitness estimates for each network. NEAT creates a new population by repeatedly calling the BREED-NET function (line 18), 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 19–20). The remainder of this section provides an overview of the reproductive process that occurs in lines 17–20. Stanley and Miikkulainen (Stanley and Miikkulainen, 2002) present a full description.

2.2.1 Minimizing Dimensionality

Unlike other systems that evolve network topologies and weights Gruau et al. (1996); Yao (1999) 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.



(a) A mutation operator for adding new nodes



(b) A mutation operator for adding new links

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

2.2.2 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. 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 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. 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 Radcliffe (1993) is essentially avoided.

2.2.3 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 ($\overline{W}$):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

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* Goldberg and Richardson (1987), where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

3. Method

This section describes evolutionary function approximation and a complete implementation called NEAT+Q. It also describes on-line evolutionary computation and details two ways of implementing it in NEAT+Q.

3.1 Evolutionary Function Approximation

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. How these effects occur depends on which of two possible approaches is employed. The first possibility is a *Lamarckian* approach, in which the changes made by TD during a given generation are written back into the original genomes, which are then used to breed a new population. The second possibility is a *Darwinian* implementation, in which the changes made by TD are discarded and the new population is bred from the original genomes, as they were at birth.

It has long since been determined that biological systems are Darwinian, not Lamarckian. However, it remains unclear which approach is better computationally, despite substantial research Pereira and Costa (2001); D. Whitley (1994); Yamasaki and Sekiguchi (2000). The potential advantage of Lamarckian evolution is obvious: it prevents each generation from having to repeat the same learning. However, Darwinian evolution can be advantageous because it enables each generation to reproduce the genomes that led to success in the previous generation, rather than relying on altered versions that may not thrive under continued alteration. Furthermore, in a Darwinian system, the learning conducted by previous generations can be indirectly recorded in a population's genomes via a phenomenon called the *Baldwin Effect* Baldwin (1896), which has been demonstrated in evolutionary computation Hinton and Nowlan (1987); Ackley and Littman (1991); Boers et al. (1995); Arita and Suzuki (2000). The Baldwin Effect occurs in two stages. In the first stage, the learning performed by individuals during their lifetimes speeds evolution, because each individual does not have to be exactly right at birth; it need only be in the right neighborhood and learning can adjust it accordingly. In the second stage, those behaviors that were previously learned during individuals' lifetimes become known at birth. This stage occurs because individuals that possess adaptive behaviors at birth have higher overall fitness and are favored by evolution.

Hence, synergistic effects between evolution and learning are possible regardless of which implementation is used. In Section 5, we compare the two approaches empirically. The remainder of this section details NEAT+Q, the implementation of evolutionary function approximation used in our experiments.

3.1.1 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 3 summarizes the resulting NEAT+Q method. Note that this algorithm is identical to Algorithm 2, except for the delineated section containing lines 13–16. Each time the agent takes an action, the network is backpropagated towards Q-learning targets (line 16) and ϵ-greedy selection occurs just as in Algorithm 1 (lines 13–14). If α and ϵ_{td} are set to zero, this method degenerates to regular NEAT.

Algorithm 3 NEAT+Q($S, A, c, p, m_n, m_l, g, e, \alpha, \gamma, \lambda, \epsilon_{td}$)

```
 1: // S: set of all states, A: set of all actions, c: output scale, 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_td: exploration rate
 5:
 6: P[] ← INIT-POPULATION(S, A, p)      // create new population P with random networks
 7: for i ← 1 to g do
 8:   for j ← 1 to e do
 9:     N, s, s' ← RANDOM(P[]), null, INIT-STATE(S)      // select a network randomly
10:     repeat
11:       Q[] ← c× EVAL-NET(N, s')      // compute value estimates for current state
12:       ----------------------------------------------------------------
13:       with-prob(ε_td) a' ← RANDOM(A)      // select random exploratory action
14:       else a' ← argmax_k Q[k]      // or select greedy action
15:       if s ≠ null then
16:         BACKPROP(N, s, a, (r + γ max_k Q[k])/c, α, γ, λ) // adjust weights toward target
17:       ----------------------------------------------------------------
18:       s, a ← s', a'
19:       r, s' ← TAKE-ACTION(a')      // take action and transition to new state
20:       N.fitness ← N.fitness + r      // update total reward accrued by N
21:     until TERMINAL-STATE?(s)
22:     N.episodes ← N.episodes + 1      // update total number of episodes for N
23:   P'[] ← new array of size p      // new array will store next generation
24:   for j ← 1 to p do
25:     P'[j] ← BREED-NET(P[])      // make a new network based on fit parents in P
26:     with-probability m_n: ADD-NODE-MUTATION(P'[j]) // add a node to new network
27:     with-probability m_l: ADD-LINK-MUTATION(P'[j])  // add a link to new network
28:   P[] ← P'[]
```

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.

In NEAT+Q, the weight changes caused by backpropagation accumulate in the current population's networks throughout each generation. When a network is selected for an episode, its weights begin exactly as they were at the end of its last episode. In the Lamarckian approach, those changes are copied back into the networks' genomes and inherited by their offspring. In the Darwinian approach, those changes are discarded at the end of each generation.

3.2 On-Line Evolutionary Computation

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. TD methods excel at on-line tasks because they are typically combined with action selection mechanisms that achieve this balance (e.g ϵ-greedy and softmax selection).

Evolutionary methods, though lacking explicit selection mechanisms, do implicitly perform this balance. In fact, in one of the earliest works on evolutionary computation, Holland (Holland, 1975) 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 members of each generation have been determined, they all typically receive the same evaluation time, even if some individuals dramatically outperform others in early episodes. Hence, within a generation, a typical evolutionary method is purely exploratory, as it makes no effort to favor those individuals that have performed well so far.

Therefore, to excel on-line, evolutionary methods need a way to limit the exploration that occurs within each generation and force more exploitation. In a sense, this problem is the opposite of that faced by TD methods, which naturally exploit (by following the greedy policy) and thus need a way to force more exploration. Nonetheless, the ultimate goal is the same: a proper balance between the two extremes. Hence, we propose that the solution can be the same too. In this section, we discuss ways of borrowing the action selection mechanisms traditionally used in TD methods and applying them in evolutionary computation.

To do so, 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 evaluations, 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, an approach we call on-line evolutionary computation. Using this technique, evolutionary algorithms can excel on-line by balancing exploration and exploitation within *and* across generations.

The remainder of this section presents two implementations. The first, which relies on ϵ-greedy selection, switches probabilistically between searching for better policies and re-evaluating the best known policy to garner maximal reward. The second, which relies on softmax selection, distributes evaluations in proportion to each individual's estimated fitness, thereby focusing on the most promising individuals and increasing the average reward accrued.

3.2.1 Using ϵ-Greedy Selection in Evolutionary Computation

When ϵ-greedy selection is used in TD methods, a single parameter, ϵ_{td}, is used to control 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_{td}, it will explore by selecting randomly from the available actions. With probability $1 - \epsilon_{td}$, it will exploit by selecting the greedy action.

In evolutionary computation, this same mechanism can be used to determine which policies to evaluate within each generation. With probability ϵ_{ec}, the algorithm explores by behaving exactly as it would normally: selecting a policy for evaluation, either randomly or by iterating through the population. With probability $1 - \epsilon_{ec}$, 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 and NEAT+Q, we need only alter the assignment of the candidate policy N in lines 7 and 9 of Algorithms 2 and 3, respectively. Instead of a random selection, we use the result of the ϵ-greedy selection function described in Algorithm 4, where $N.average = N.fitness/N.episodes$. In the case of NEAT+Q, two different ϵ parameters control exploration throughout the system: ϵ_{td} controls the exploration that helps Q-learning estimate the value function and ϵ_{ec} controls exploration that helps NEAT discover appropriate topologies and initial weights for the neural network function approximators.

Algorithm 4 ϵ-GREEDY SELECTION(P, ϵ_{ec})

```
// P: population, ε_ec: NEAT's exploration rate

with-prob(ε_ec) return RANDOM(P)                          // select random network
else return N ∈ P | ∀(N′ ∈ P) N.average ≥ N′.average     // or select champion
```

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 evolutionary computation to intelligently focus search with each generation and create a more nuanced balance between exploration and exploitation.

3.2.2 Using Softmax Selection in Evolutionary Computation

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 Sutton and Barto (1998), in which case an agent in state s chooses an action a with probability

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

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 evolutionary computation 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

$$\frac{e^{S(p)/\tau}}{\sum_{q \in P} e^{S(q)/\tau}}$$

where $S(p)$ is the average fitness of the policy p.

To apply softmax selection to NEAT and NEAT+Q, we need only alter the assignment of the candidate policy N in lines 7 and 9 of Algorithms 2 and 3, respectively. Instead of a random selection, we use the result of the softmax selection function shown in Algorithm 5. In the case of NEAT+Q, ϵ_{td} controls Q-learning's exploration and τ controls NEAT's exploration. Of course, softmax exploration could be used within Q-learning too. However, since comparing different selection mechanisms for TD methods is not the subject of our research, in this paper we use only ϵ-greedy selection with TD methods.

Algorithm 5 SOFTMAX SELECTION(P, τ)

// *P: population, τ: softmax temperature*

if $\exists N \in P \mid N.episodes = 0$ **then**
  return N // *give each network one episode before using softmax*
else
  $total \leftarrow \sum_{N \in P} e^{N.average/\tau}$ // *compute denominator of Boltzmann function*
  for all $N \in P$ **do**
    with-prob($\frac{e^{N.average/\tau}}{total}$) return N // *select N for evaluation*
    else $total \leftarrow total - e^{N.average/\tau}$ // *or skip N and reweight probabilities*

In addition to providing a more nuanced balance between exploration and exploitation, softmax selection also allows evolutionary computation to more effectively focus its search within each generation. Instead of spending the same number of evaluations on each member of the population, softmax selection can quickly abandon poorly performing policies and spend more episodes evaluating the most promising individuals.

In summary, on-line evolutionary computation enables the use of evolutionary computation during an agent's interaction with the world. Therefore, the ability of evolutionary function approximation to optimize representations need not come at the expense of the on-line aspects of TD methods. On the contrary, the value function and its representation can be optimized simultaneously, all while the agent interacts with its environment.

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 benchmark task requiring function approximation. We use this domain to establish preliminary, proof-of-concept results for the novel methods described in this paper. The second domain, server job scheduling, is a large, probabilistic domain drawn from the field of autonomic computing. We use this domain to assess whether these new methods can scale to a much more complex task. The remainder of this section details each of these domains and describes our approach to solving them with reinforcement learning.

4.1 Mountain Car

In the mountain car task Boyan and Moore (1995), 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.

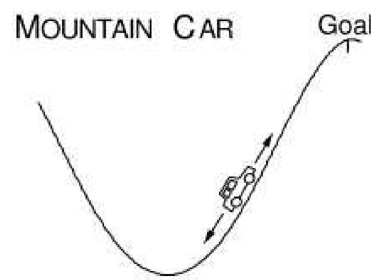


Figure 2: The Mountain Car Task. This figure was taken from Sutton and Barto (Sutton and Barto, 1998).

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.

Though the agent's state has only two features, they are continuous and hence learning the value function requires a function approximator. Previous research has demonstrated that TD methods can solve the mountain car task using several different function approximators, including CMACs Sutton (1996); Kretchmar and Anderson (1997), locally weighted regression Boyan and Moore (1995), decision trees Pyeatt and Howe (2001), radial basis functions Kretchmar and Anderson (1997), and instance-based methods Boyan and Moore (1995). By giving the learner *a priori* knowledge about the goal state and using methods based on experience replay, the mountain car problem has been solved with neural networks too Reidmiller (2005). However, the task remains notoriously difficult for neural networks, as several researchers have noted that value estimates can easily diverge Boyan and Moore (1995); Pyeatt and Howe (2001).

We hypothesized that the difficulty of using neural networks in this task is due at least in part to the problem of finding an appropriate representation. Hence, as a preliminary evaluation of evolutionary function approximation, we applied NEAT+Q to the mountain car task to see if it could learn better than manually designed networks. The results are presented in Section 5.

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 our methods can scale to 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 Kephart and Chess (2003). 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. The demand for such features is growing rapidly, since computer systems are becoming so complex that maintaining them with human support staff is increasingly infeasible.

The vision of autonomic computing poses new challenges to many areas of computer science, including architecture, operating systems, security, and human-computer interfaces. However, the burden on artificial intelligence is especially great, since intelligence is a prerequisite for self-managing systems. In particular, we believe machine learning will play a primary role, since computer systems must be adaptive if they are to perform well autonomously. There are many ways to apply supervised methods to autonomic systems, e.g. for intrusion detection Ertoz et al. (2004), spam filtering Dalvi et al. (2004), or system configuration Wildstrom et al. (2005). However, there are also many tasks where no human expert is available and reinforcement learning is applicable, e.g network routing Boyan and Littman (1994), job scheduling Whiteson and Stone (2004), and cache allocation Gomez et al. (2001).

One such task is server job scheduling, 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* (not to be confused with a TD value function) for each job type maps the job's completion time to the utility derived by the user Walsh et al. (2004). The problem of server job scheduling becomes challenging when these utility functions are nonlinear 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. Also, this domain is challenging because it is large (the size of both the state and action spaces grow in direct proportion to the size of the queue) and probabilistic (the server does not know what type of job will arrive next). Hence, it is a typical example of a reinforcement learning task that requires effective function approximation.

The server job scheduling task is quite different from traditional scheduling tasks Zhang and Dietterich (1995); Zweben and Fox (1998). In the latter case, there are typically multiple resources available and each job has a partially ordered list of resource requirements. Server job scheduling is simpler because there is only one resource (the server) and all jobs are independent of each other. However, it is more complex in that performance is measured via arbitrary utility functions, whereas traditional scheduling tasks aim solely to minimize completion times.

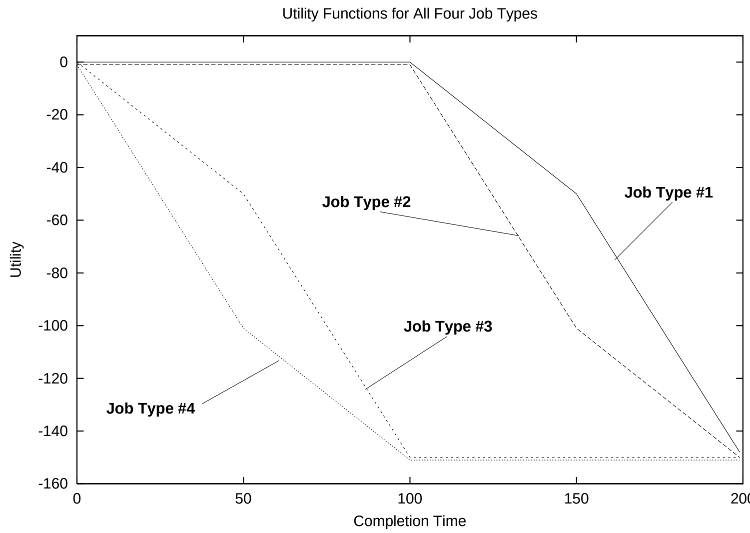


Figure 3: The four utility functions used in our experiments.

Our experiments were conducted in a Java-based simulator. The simulation begins with 100 jobs preloaded into the server's queue and ends when the queue becomes empty. 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. 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.

Four different job types were used in our experiments. Hence, the task can generate 4^{200} unique episodes. Utility functions for the four job types 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.

A primary obstacle to applying reinforcement learning methods to this domain is the size of the state and action spaces. A complete state description includes the type and age of each job in the queue. The scheduler's actions consist of selecting jobs for processing; hence a complete action space includes every job in the queue. To render these 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 processes the youngest job in the queue that matches the type and age range specified by the action.

These discretizations mean the agent has less information about the contents of the job queue. However, its state is still sufficiently detailed to allow effective learning. Although the utility functions can change dramatically within each age range, their slopes do not change. It is the slope of the utility function, not the utility function itself, which determines how much utility is lost by delaying a given job.

Even after discretization, the state space is quite large. If the queue holds at most q_{max} jobs, $\binom{q_{max}+1}{16}$ is a loose upper bound on the number of states, since each job can be in one of 16 buckets. Some of these states will not occur (e.g. ones where all the jobs in the queue are in the youngest age range). Nonetheless, with 16 actions per state, it is clearly infeasible to represent the value function in a table. Hence, success in this domain requires function approximation, as addressed in the following section.

5. Results

We conducted a series of experiments in the mountain car and server job scheduling domains to empirically evaluate the methods presented in this paper. Section 5.1 compares manual and evolutionary function approximators. Section 5.2 compares off-line and on-line evolutionary computation. Section 5.3 tests evolutionary function approximation combined with on-line evolutionary computation. Section 5.4 compares these novel approaches to previous learning and non-learning methods. Section 5.5 compares Darwinian and Lamarckian versions of evolutionary function approximation. Finally, Section 5.6 presents some addition tests that measure the effect of continual learning on function approximators. The results offer insight into why certain methods outperform others in these domains and what factors can make neural network function approximation difficult in practice.

Each of the graphs presented in these sections include error bars indicating 95% confidence intervals. In addition, to assess statistical significance, we conducted Student's t-tests on each pair of methods evaluated. The results of these tests are summarized in Appendix A.

5.1 Comparing Manual and Evolutionary Function Approximation

As an initial baseline, we conducted, in each domain, 25 runs in which NEAT attempts to discover a good policy using the setup described in Section 4. In these runs, the population size p was 100, the number of generations g was 100, the node mutation rate m_n was 0.02, the link mutation rate m_l was 0.1, and the number of episodes per generation e was 10,000. Hence, each individual was evaluated for 100 episodes on average. See Appendix B for more details on the NEAT parameters used in our experiments.

Next, we performed 25 runs in each domain using NEAT+Q, with the same parameter settings. The eligibility decay rate λ was 0.0. and the learning rate α was set to 0.1 and annealed linearly for each member of the population until reaching zero after 100 episodes.[2]

In scheduling, γ was 0.95 and ϵ_{td} was 0.05. Those values of γ and ϵ_{td} work well in mountain car too, though in the experiments presented here they were set to 1.0 and 0.0 respectively, since Sutton (Sutton, 1996) found that discounting and exploration are unnecessary in mountain car. The output scale c was set to -100 in mountain car and -1000 in scheduling.

We tested both Darwinian and Lamarckian NEAT+Q in this manner. Both perform well, though which is preferable appears to be domain dependent. For simplicity, in this section and those that follow, we present results only for Darwinian NEAT+Q. In Section 5.5 we present a comparison of the two approaches.

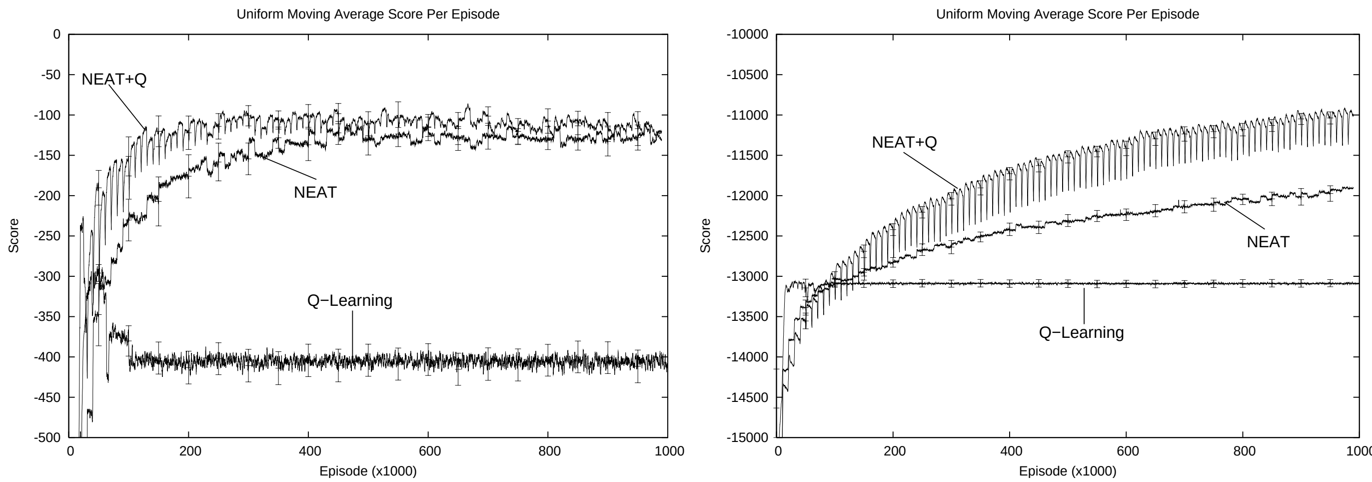
To test Q-learning without NEAT, we tried 24 different configurations in each domain. These configurations correspond to every possible combination of the following parameter settings. The networks had feed-forward topologies with 0, 4, or 8 hidden nodes. The learning rate α was either 0.01 or 0.001. The annealing schedules for α were linear, decaying to zero after either 100,000 or 250,000 episodes. The eligibility decay rate λ was either 0.0 or 0.6. The other parameters, γ and ϵ, were set just as with NEAT+Q, and the standard deviation of initial weights σ was 0.1. Each of these 24 configurations was evaluated for 5 runs. In addition, we experimented informally with higher and lower values of α, higher values of γ, slower linear annealing, exponential annealing, and no annealing at all, though none performed as well as the results presented here.

In these experiments, each run used a different set of initial weights. Hence, the resulting performance of each configuration, by averaging over different initial weight settings, does not account for the possibility that some weight settings perform consistently better than others. To address this, for each domain, we took the best performing configuration[3] and randomly selected five fixed initial weight settings. For each setting, we conducted 5 additional runs. Finally, we took the setting with the highest performance and conducted an additional 20 runs, for a total of 25. For simplicity, the graphs that follow show only this Q-learning result: the best configuration with the best initial weight setting.

Figure 4 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. Using average performance, as we do throughout this paper, is somewhat unorthodox for evolutionary methods, which are more commonly evaluated on the performance of the generation champion. There are two reasons why we adopt average performance. First, it creates a consistent metric for all the methods tested, including the TD methods that do not use evolutionary computation and hence have no generation champions. Second, it is an on-line metric because it incorporates *all* the reward the learning system accrues. Plotting only generation champions is an implicitly off-line metric because it does not penalize methods that discover good policies but fail to accrue much reward while learning. Hence, average reward is a better metric for evaluating on-line evolutionary computation, as we do in Section 5.2.

To make a larger number of runs computationally feasible, both NEAT and NEAT+Q were run for only 100 generations. In the scheduling domain, neither method has completely plateaued by this point. However, a handful of trials conducted for 200 generations verified

2. Other values of λ were tested in the context of NEAT+Q but had little effect on performance.
3. Mountain car parameters were: 4 hidden nodes, $\alpha = 0.001$, annealed to zero at episode 100,000, $\lambda = 0.0$. Server job scheduling parameters were: 4 hidden nodes, $\alpha = 0.01$, annealed to zero at episode 100,000, $\lambda = 0.6$.



(a) Mountain Car (b) Server Job Scheduling

Figure 4: A comparison of the performance of manual and evolutionary function approximators in the mountain car and server job scheduling domains.

that only very small additional improvements are made after 100 generation, without a qualitative effect on the results.

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. Although each individual learns for only 100 episodes on average, NEAT's system of randomly selecting individuals for evaluation causes that learning to be spread across the entire generation: 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 we tested and network configurations we tried, evolutionary function approximation significantly improves performance over manually designed networks. In the scheduling domain, Q-learning learns much more rapidly in the very early part of learning. In both domains, however, Q-learning soon plateaus while NEAT and NEAT+Q continue to improve. Of course, after 100,000 episodes, Q-learning's learning rate α has annealed to zero and no additional learning is possible. However, its performance plateaus well before α reaches zero and, in our experiments, running Q-learning with slower annealing or no annealing at all consistently led to inferior and unstable performance.

Nonetheless, the possibility remains that additional engineering of the network structure, the feature set, or the learning parameters would significantly improve Q-learning's performance. In particular, when Q-learning is started with one of the best networks discovered by NEAT+Q and the learning rate is annealed aggressively, Q-learning matches NEAT+Q's performance without directly using evolutionary computation. However, it is unlikely that a manual search, no matter how extensive, would discover these successful topologies, which contain irregular and partially connected hidden layers. Figure 5 shows examples of typical networks evolved by NEAT+Q.

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,

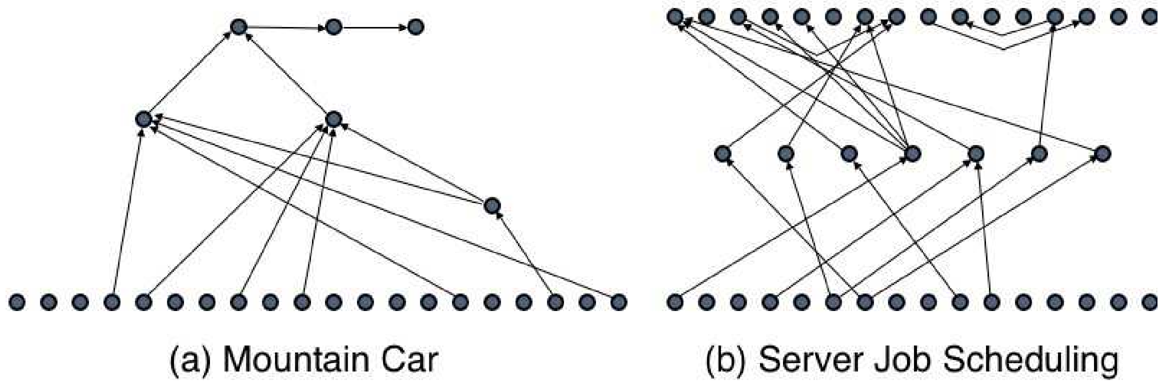


Figure 5: Typical examples of the topologies of the best networks evolved by NEAT+Q in both the mountain car and scheduling domains. Input nodes are on the bottom, hidden nodes in the middle, and output nodes on top. In addition to the links shown, each input node is directly connected to each output node. Note that two output nodes can be directly connected, in which case the activation of one node serves not only as an output of the network, but as an input to the other node.

though both plateau at approximately the same level. In the server job scheduling domain, NEAT+Q learns more rapidly and also converges to significantly 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.

5.2 Comparing Off-Line and On-Line Evolutionary Computation

In this section, we present experiments evaluating on-line evolutionary computation. Since on-line evolutionary computation does not depend on evolutionary function approximation, we first test it using regular NEAT, by comparing an off-line version to on-line versions using ϵ-greedy and softmax selection. In Section 5.3 we study the effect of combining NEAT+Q with on-line evolutionary computation.

Figure 6 compares the performance of off-line NEAT to its on-line counterparts in both domains. The results for off-line NEAT are the same as those presented in Figure 4. To test on-line NEAT with ϵ-greedy selection, 25 runs were conducted with ϵ_{ec} 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. To test on-line NEAT with softmax selection, 25 runs were conducted with τ set to 50 in mountain car and 500 in the scheduling domain. These values are different because a good value of τ depends on the range of possible values, which differ dramatically between the two domains.

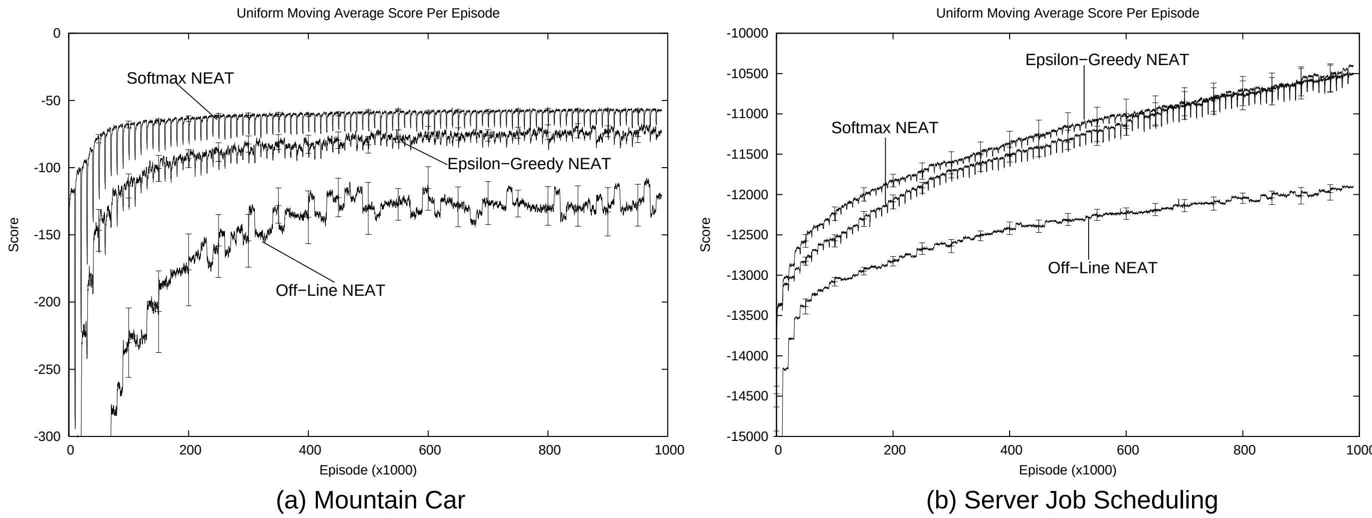


Figure 6: A comparison of the performance off-line and on-line evolutionary computation in the mountain car and server job scheduling domains.

These results demonstrate that both versions of on-line evolutionary computation can significantly improve NEAT's average performance. In addition, in mountain car, on-line evolutionary computation with softmax selection boosts performance even more than ϵ-greedy selection.

Given the way these two methods work, the advantage of softmax over ϵ-greedy in mountain car is not surprising. ϵ-greedy selection is a rather naïve approach because it treats all exploratory actions equally, with no attempt to favor the most promising ones. For the most part, it conducts the search for better policies in the same way as off-line evolutionary computation; it simply interleaves that search with exploitative episodes that employ the best known policy. Softmax selection, by contrast, concentrates exploration on the most promising alternatives and hence alters the way the search for better policies is conducted. Unlike ϵ-greedy exploration, softmax selection spends fewer episodes on poorly performing individuals and more on those with the most promise. In this way, it achieves better performance.

More surprising is that this effect is not replicated in the scheduling domain. Both on-line methods perform significantly better than their off-line counterpart but softmax performs only as well as ϵ-greedy. It is possible that softmax, though focusing exploration more intelligently, exploits less aggressively than ϵ-greedy, which gives so many evaluations to the champion. It is also possible that some other setting of τ would make softmax outperform ϵ-greedy, though our informal parameter search did not uncover one. Even achieving the performance shown here required using different values of τ in the two domains, whereas the same value of ϵ worked in both cases. This highlights one disadvantage of using softmax selection: the difficulty of choosing τ. 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." Sutton and Barto (1998, pages 27-30)

It is interesting that the intra-generational learning curves characteristic of NEAT+Q appear in the on-line methods even though backpropagation is not used. The average performance increases during each generation without the help of TD methods because the system becomes better informed about which individuals to select on exploitative episodes. Hence, on-line evolutionary computation can be thought of as another way of combining evolution and learning. In each generation, the system learns which members of the population are strongest and uses that knowledge to boost average performance.

5.3 Combining Evolutionary Function Approximation with On-Line Evolutionary Computation

Sections 5.1 and 5.2 verify that both evolutionary function approximation and on-line evolutionary computation can significantly boost performance in reinforcement learning tasks. In this section, we present experiments that assess how well these two ideas work together.

Figure 7 presents the results of combining NEAT+Q with softmax evolutionary computation, averaged over 25 runs, and compares it to using each of these methods individually, i.e. using off-line NEAT+Q (as done in Section 5.1) and using softmax evolutionary computation with regular NEAT (as done in Section 5.2). For the sake of simplicity we do not present results for ϵ-greedy NEAT+Q though we tested it and found that it performed similarly to softmax NEAT+Q.

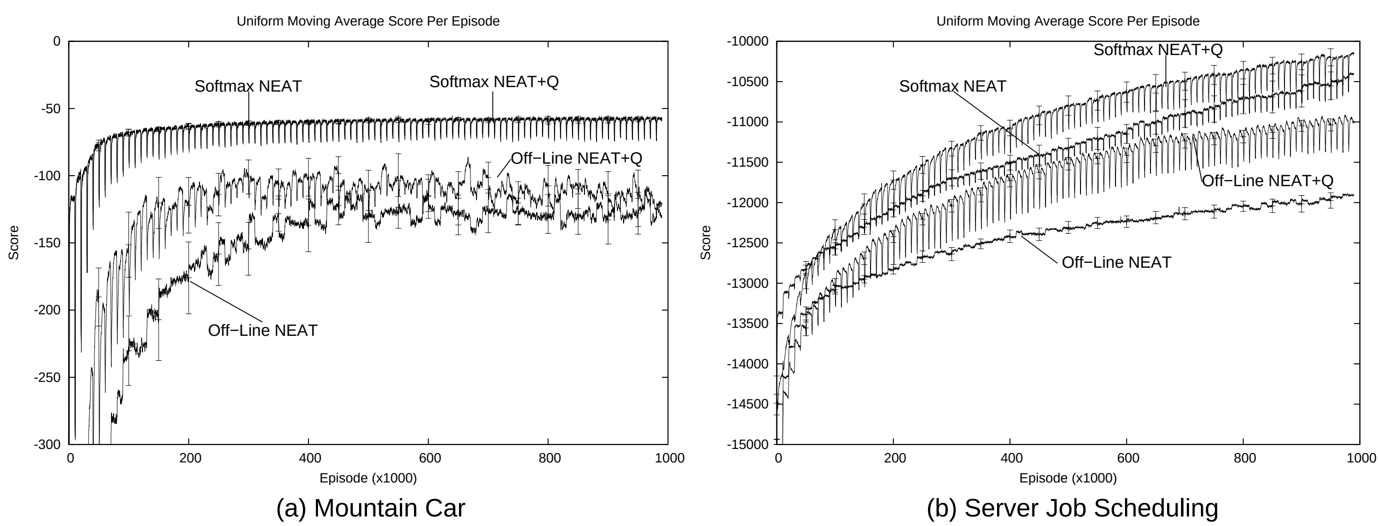


Figure 7: The performance of combining evolutionary function approximation with on-line evolutionary computation compared to using each individually in the mountain car and server job scheduling domains.

In both domains, softmax NEAT+Q performs significantly better than off-line NEAT+Q. Hence, just like regular evolutionary computation, evolutionary function approximation performs better when supplemented with selection techniques traditionally used in TD methods. Surprisingly, in the mountain car domain, softmax NEAT+Q performs only as well softmax NEAT. We attribute these results to a ceiling effect, i.e. the mountain car domain is easy enough that, given an appropriate selection mechanism, NEAT is able to learn quite rapidly, even without the help of Q-learning. In the server job scheduling domain, softmax NEAT+Q does perform better than softmax NEAT, though the difference is rather modest. Hence, in both domains, the most critical factor to boosting the performance of evolutionary computation is the use of an appropriate selection mechanism.

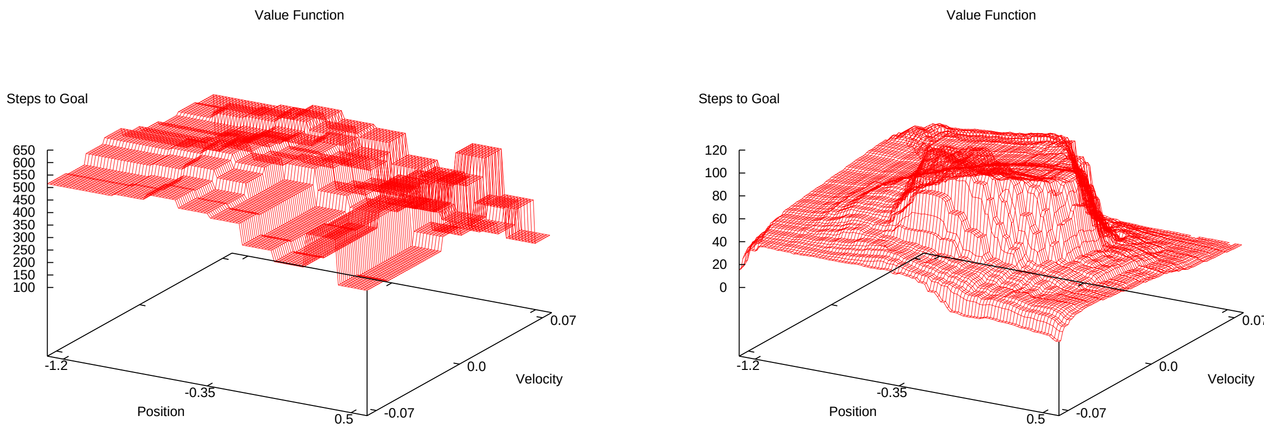
5.4 Comparing to Previous Approaches

The experiments presented thus far verify that the novel methods presented in this paper can improve performance over the constituent techniques upon which they are built. In this section, we present experiments that compare the performance of the highest performing novel method, softmax NEAT+Q, to previous approaches. In the mountain car domain, we compare to previous results that use TD methods with a linear function approximator Sutton (1996). In the server job scheduling domain, we compare to a random scheduler, two non-learning schedulers from previous research van Mieghem (1995); Whiteson and Stone (2004), and an analytical solution computed using integer linear programming.

In the mountain car domain, the results presented above make clear that softmax NEAT+Q can rapidly learn a good policy. However, since these results use an on-line metric, performance is averaged over all members of the population. Hence, they do not reveal how close the best learned policies are to optimal. To assess this, we selected the generation champion from the final generation of each softmax NEAT+Q run and evaluated it for an additional 1,000 episodes. Then we compared this to the performance of a learner using Sarsa, a TD method similar to Q-learning Sutton and Barto (1998), with CMACs, a popular linear function approximator Sutton and Barto (1998), using a setup that matches that of Sutton (Sutton, 1996) as closely as possible. We found their performance to be nearly identical: softmax NEAT+Q received an average score of -52.75 while the Sarsa CMAC learner received -52.02. We believe this performance is approximately optimal, as it matches the best results published by other researchers, e.g. Smart and Kaelbling (2000).

This does not imply that neural networks are the function approximator of choice for the mountain car domain. On the contrary, Sutton's CMACs converge in many fewer episodes. Nonetheless, these results demonstrate that evolutionary function approximation and on-line evolution make it feasible to find approximately optimal policies using neural networks, something that some previous approaches Boyan and Moore (1995); Pyeatt and Howe (2001), using manually designed networks, were unable to do.

Since the mountain car domain has only two state features, it is possible to visualize the value function. Figure 8 compares the value functions learned by softmax NEAT+Q to that of Sarsa with CMACs. For clarity, the graphs plot estimated steps to the goal. Since the agent receives a reward of -1 for each timestep until reaching the goal, this is equivalent to $-\max_a(Q(s,a))$. Surprisingly, the two value functions bear little resemblance to one another. While they share some very general characteristics, they differ markedly in both shape and scale. Hence, these graphs highlight a fact that has been noted before Tesauro (1994): that TD methods can learn excellent policies even if they estimate the value function only very grossly. So long as the value function assigns the highest value to the correct action, the agent will perform well.



(a) NEAT+Q Network

(b) CMAC

Figure 8: The value function, shown as estimated steps to the goal, of policies learned by softmax NEAT+Q and Sarsa using CMACs.

In the server job scheduling domain, finding alternative approaches for comparison is less straightforward. Substantial research about job scheduling already exists but most of the methods involved are not applicable here because they do not allow jobs to be associated with arbitrary utility functions. For example, Liu and Layland (Liu and Layland, 1973) present methods for job scheduling in a real-time environment, in which a hard deadline is associated with each job. McWherter et al. (McWherter et al., 2004) present methods for scheduling jobs with different priority classes. However, unlike the utility functions shown in Section 4.2, the relative importance of a job type does not change as a function of time. McGovern et al. (McGovern et al., 2002) use reinforcement learning for CPU instruction scheduling but aim only to minimize completion time.

One method that can be adapted to the server job scheduling task is the generalized $c\mu$ rule van Mieghem (1995), in which the server always processes at time t the oldest job of that type k which maximizes $C'_k(o_k)/p_k$, where C'_k is the derivative of the cost function for job type k, o_k is the age of the oldest job of type k and p_k is the average processing time for jobs of type k. Since in our simulation all jobs require unit time to process and the cost function is just the additive inverse of the utility function, this is equivalent to processing the oldest job of that type k that maximizes $-U'_k(o_k)$, where U'_k is the derivative of the utility function for job type k. The generalized $c\mu$ rule has been proven approximately optimal given convex cost functions van Mieghem (1995). Since the utility functions, and hence the cost functions, are both convex and concave in our simulation, there is no theoretical guarantee about its performance in the server job scheduling domain. To see how well it performs in practice, we implemented it in our simulator and ran it for 1,000 episodes, obtaining an average score of -10,891.

Another scheduling algorithm applicable to this domain is the insertion scheduler, which performed the best in a previous study of a very similar domain Whiteson and Stone (2004).

The insertion scheduler uses a simple, fast heuristic: it always selects for processing the job at the head of the queue but it keeps the queue ordered in a way it hopes will maximize aggregate utility. For any given ordering of a set of J jobs, the aggregate utility is:

$$\sum_{i \in J} U_i(a_i + p_i)$$

where $U_i(\cdot)$, a_i, and p_i are the utility function, current age, and position in the queue, respectively, of job i. Since there are $|J|!$ ways to order the queue, it is clearly infeasible to try them all. Instead, the insertion scheduler uses the following simple, fast heuristic: every time a new job is created, the insertion scheduler tries inserting it into each position in the queue, settling on whichever position yields the highest aggregate utility. Hence, by bootstrapping off the previous ordering, the insertion scheduler must consider only $|J|$ orderings. We implemented the insertion scheduler in our simulator and ran it for 1,000 episodes, obtaining an average score of -13,607.

Neither the $c\mu$ rule nor the insertion scheduler perform as well as softmax NEAT+Q, whose final generation champions received an average score of -9,723 over 1,000 episodes. Softmax NEAT+Q performed better despite the fact that the alternatives rely on much greater *a priori* knowledge about the dynamics of the system. Both alternatives require the scheduler to have a predictive model of the system, since their calculations depend on knowledge of the utility functions and the amount of time each job takes to complete. By contrast, softmax NEAT+Q, like many reinforcement learning algorithms, assumes such information is hidden and discovers a good policy from experience, just by observing state transitions and rewards.

If, in addition to assuming the scheduler has a model of the system, we make the unrealistic assumption that unlimited computation is available to the scheduler, then we can obtain an informative upper bound on performance. At each time step of the simulation, we can compute the optimal action analytically by treating the scheduling problem as an integer linear program. For each job $i \in J$ and for each position j in which it could be placed, the linear program contains a variable $x_{ij} \in \{0, 1\}$. Associated with each variable is a weight $w_{ij} = U_i(a_i + j)$, which represents the reward the scheduler will receive when job i completes given that it currently resides in position j. Since the scheduler's goal is to maximize aggregate utility, the linear program must maximize $\sum_i \sum_j w_{ij} x_{ij}$. In addition to the constraint that $\forall ij : x_{ij} \in \{0, 1\}$, the program is also constrained such that each job is in exactly one position: $\forall i : \sum_j x_{ij} = 1$ and that each position holds exactly one job: $\forall j : \sum_i x_{ij} = 1$.

A solution to the resulting integer linear program is an ordering that will maximize the aggregate utility of the jobs currently in the queue. If the scheduler always processes the job in the first position of this ordering, it will behave optimally *assuming no more jobs arrive*. Since new jobs are constantly arriving, the linear program must be re-solved anew at each time step. The resulting behavior may still be suboptimal since the decision about which job to process is made without reasoning about what types of jobs are likely to arrive later. Nonetheless, this analytical solution represents an approximate upper bound on performance in this domain.

Using the CPLEX software package, we implemented a scheduler based on the linear program described above and tested in our simulator for 1,000 episodes, obtaining an average score of -7,819. Not surprisingly, this performance is superior to that of softmax NEAT+Q, though it takes, on average, 741 times as long to run. The computational requirements of this solution are not likely to scale well either, since the number of variables in the linear program grows quadratically with respect to the size of the queue.

Figure 9 summarizes the performance of the alternative scheduling methods described in this section and compares them to softmax NEAT+Q. It also includes, as a lower bound on performance, a random scheduler, which received an average score of -15,502 over 1,000 episodes. A Student's t-test verified that the difference in performance between each pair of methods is statistically significant with 95% confidence. Softmax NEAT+Q performs the best except for the linear programming approach, which is computationally expensive and relies on a model of the system. Prior to learning, softmax NEAT+Q performs similarly to the random scheduler. The difference in performance between the best learned policies and the linear programming upper bound is 75% better than that of the baseline random scheduler and 38% better than that of the next best method, the $c\mu$ scheduler.

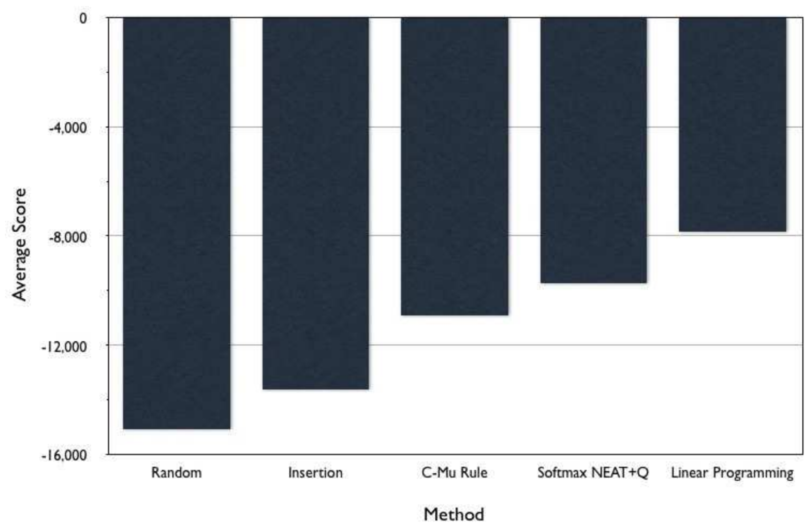


Figure 9: A comparison of the performance of softmax NEAT+Q and several alternative methods in the server job scheduling domain.

5.5 Comparing Darwinian and Lamarckian Evolutionary Computation

As described in Section 3.1, evolutionary function approximation can be implemented in either a Darwinian or Lamarckian fashion. The results presented so far all use the Darwinian implementation of NEAT+Q. However, it is not clear that this approach is superior even though it more closely matches biological systems. In this section, we compare the two approaches empirically in both the mountain car and server job scheduling domains. Many other empirical comparisons of Darwinian and Lamarckian systems have been conducted previously D. Whitley (1994); Yamasaki and Sekiguchi (2000); Pereira and Costa (2001) but ours is novel in that individual learning is based on a TD function approximator. In other words, these experiments address the question: when trying to approximate a TD value function, is a Darwinian or Lamarckian approach superior?

Figure 10 compares the performance of Darwinian and Lamarckian NEAT+Q in both the mountain car and server job scheduling domains. In both cases, we use off-line NEAT+Q, as the on-line versions tend to mute the differences between the two implementations. Though both implementations perform well in both domains, Lamarckian NEAT+Q does better in mountain car but worse in server job scheduling. Hence, the relative performance of these two approaches seems to depend critically on the dynamics of the domain to which they are applied. In the following section, we present some additional results that elucidate which factors affect their performance.

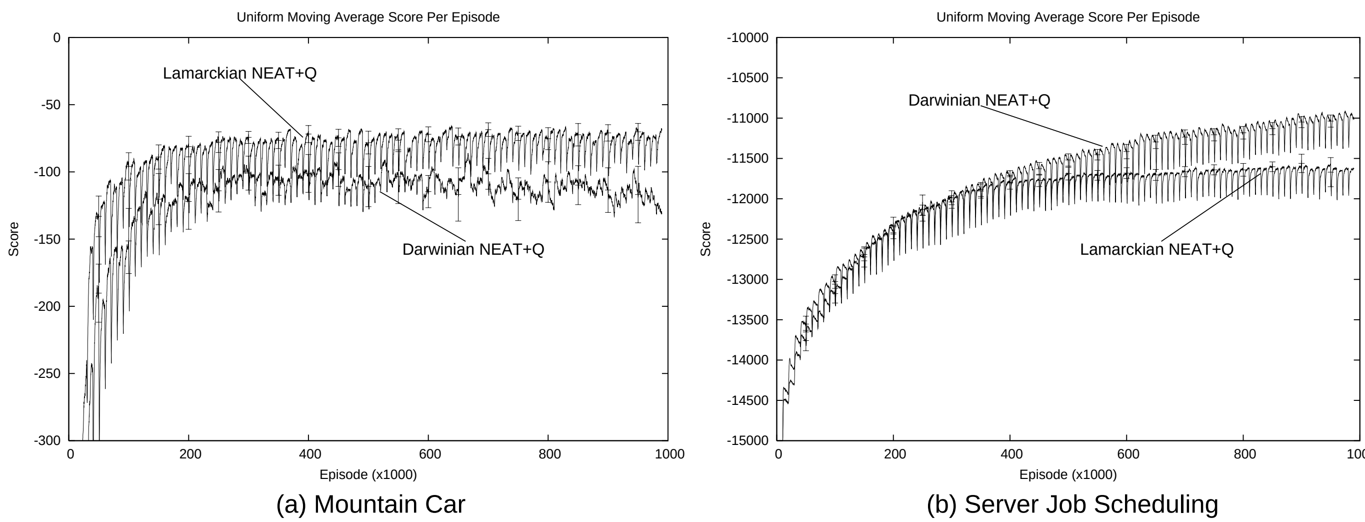


Figure 10: A comparison of Darwinian and Lamarckian NEAT+Q in the mountain car and server job scheduling domains.

5.6 Continual Learning Tests

In this section, we assess the performance of the best networks discovered by NEAT+Q when evaluated for many additional episodes. We compare two scenarios, one where the learning rate is annealed to zero after 100 episodes, just as in training, and one where it is not annealed at all. Comparing performance in these two scenarios allows us to assess the effect of continual learning on the evolved networks.

We hypothesized that NEAT+Q's best networks would perform well under continual learning in the mountain car domain but not in server job scheduling. This hypothesis was motivated by the results of early experiments with NEAT+Q. Originally, we did not anneal α at all. This setup worked fine in the mountain car domain but in scheduling it worked only with off-line NEAT+Q; on-line NEAT+Q actually performed worse than off-line NEAT+Q! Annealing NEAT+Q's learning rate eliminated the problem, as the experiments in Section 5.2 verify. If finding weights that remain stable under continual learning is more difficult in scheduling than in mountain car, it could explain this phenomenon, since ϵ-greedy and softmax selection, by giving many more episodes of learning to certain networks, could cause those networks to become unstable and perform poorly.

To test the best networks without continual learning, we selected the final generation champion from each run of off-line Darwinian NEAT+Q and evaluated it for an additional 5,000 episodes, i.e. 50 times as many episodes as it saw in training. During these additional episodes, the learning rate was annealed to zero by episode 100, just as in training. To test the best networks with continual learning, we repeated this experiment but did not anneal the learning rate at all. To prevent any unnecessary discrepancies between training and testing, we repeated the original NEAT+Q runs with annealing turned off and used the resulting final generation champions.

Figure 11 shows the results of these tests. In the mountain car domain, performance remains relatively stable regardless of whether the networks continue to learn. The networks tested without annealing show more fluctuation but maintain performance similar to those that were annealed. However, in the scheduling domain, the networks subjected to continual learning rapidly plummet in performance whereas those that are annealed continue to perform as they did in training. These results directly confirm our hypothesis that evolutionary computation can find weights that perform well under continual learning in mountain car but not in scheduling. This explains why on-line NEAT+Q does not require an annealed learning rate in mountain car but does in scheduling.

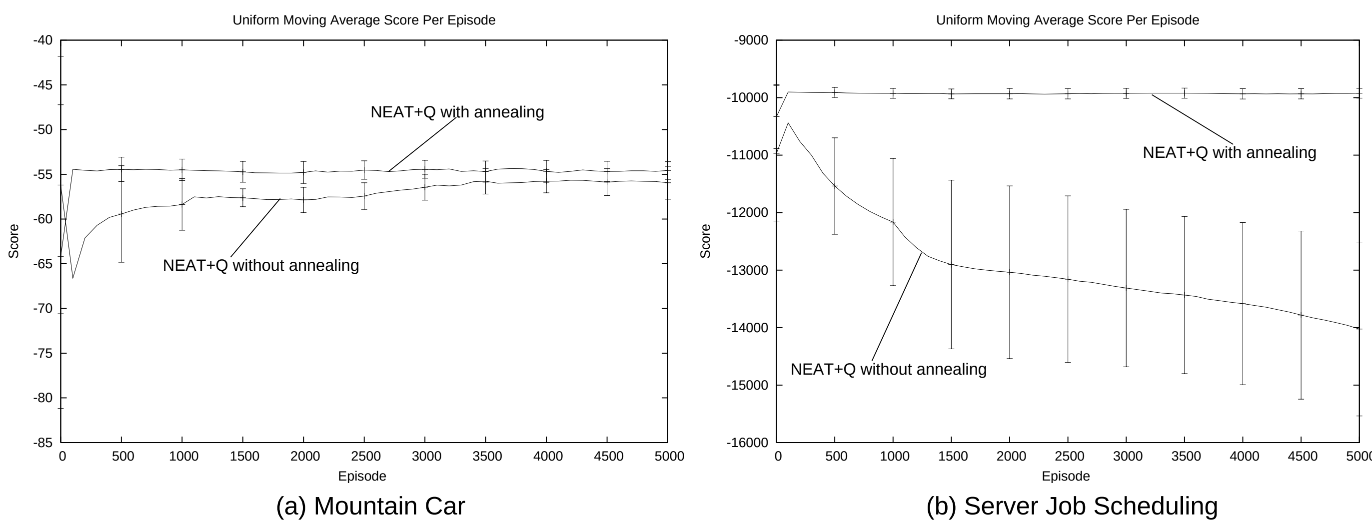


Figure 11: A comparison of the performance of the best networks evolved by NEAT+Q when tested, with and without annealing, for an additional 5,000 episodes.

These tests also shed light on the comparison between Darwinian and Lamarckian NEAT+Q presented in Section 5.5. A surprising feature of the Darwinian approach is that it is insensitive to the issue of continual learning. Since weight changes do not affect offspring, evolution need only find weights that remain suitable during one individual's lifetime. By contrast, in the Lamarckian approach, weight changes accumulate from generation to generation. Hence, the TD updates that helped in early episodes can hurt later on. In this light it makes perfect sense that Lamarckian NEAT+Q performs better in mountain car than in scheduling, where continual learning is problematic.

These results suggest that the problem of stability under continual learning can greatly exacerbate the difficulty of performing neural network function approximation in practice. This issue is not specific to NEAT+Q, since Q-learning with manually designed networks achieved decent performance only when the learning rate was properly annealed. Darwinian NEAT+Q is a novel way of coping with this problem, since it obviates the need for long-term stability. In on-line evolutionary computation annealing may still be necessary but it is less critical to set the rate of decay precisely. When learning ends, it prevents only a given individual from continuing to improve. The system as a whole can still progress, as evolution exerts selective pressure and learning begins anew in the next generation.

6. Discussion

The results in the mountain car domain presented in Section 5, demonstrate that NEAT+Q can successfully train neural network function approximators in a domain which is notoriously problematic for them. However, NEAT+Q requires many more episodes to find good solutions (by several orders of magnitude) than CMACs do in the same domain. This contrast highlights an important drawback of NEAT+Q: since each candidate network must be trained long enough to let Q-learning work, it has very high sample complexity. In ongoing research, we are investigating ways of making NEAT+Q more sample-efficient. For example, preliminary results suggest that, by pre-training networks using methods based on experience replay Lin (1992), NEAT+Q's sample complexity can be dramatically reduced.

It is not surprising that NEAT+Q takes longer to learn than CMACs because it is actually solving a more challenging problem. CMACs, like other linear function approximators, require the human designer to engineer a state representation in which the optimal value function is linear with respect to those state features (or can be reasonably approximated as such). For example, when CMACs were applied to the mountain car domain, the two state features were tiled conjunctively Sutton (1996). By contrast, nonlinear function approximators like neural networks can take a simpler state representation and *learn* the important nonlinear relationships. Note that the state representation used by NEAT+Q, while discretized, does not include any conjunctive features of the original two state features. The important conjunctive features are represented by hidden nodes that are evolved automatically by NEAT.

Conjunctively tiling all state features is feasible in mountain car but quickly becomes impractical in domains with more state features. For example, doing so in the scheduling domain would require 16 CMACs, one for each action. In addition, each CMAC would have multiple 16-dimensional tilings. If 10 tilings were used and each state feature were discretized into 10 buckets, the resulting function approximator would have $16 \times 10 \times 10^{16}$ cells. Conjunctively tiling only some state features is feasible only with a large amount of domain expertise. Hence, methods like NEAT+Q that automatically learn nonlinear representations promise to be of great practical importance.

The results in the scheduling domain demonstrate that the proposed methods scale to a much larger, probabilistic domain and can learn schedulers that outperform existing non-learning approaches. The difference in performance between the best learned policies and the linear programming upper bound is 75% better than that of the baseline random scheduler and 38% better than that of the next best method, the $c\mu$ scheduler. However, the results also demonstrate that non-learning methods can do quite well in this domain. If so, is it worth the trouble of learning? We believe so. In a real system, the utility functions that the learner maximizes would likely be drawn directly from Service Level Agreements (SLAs), which are legally binding contracts governing how much clients pay their service providers as a function of the quality of service they receive Walsh et al. (2004). Hence, even small improvements in system performance can significantly affect the service provider's bottom line. Substantial improvements like those demonstrated in our results, if replicated in real systems, could be very valuable indeed.

Overall, the main limitation of the results presented in this paper is that they apply only to neural networks. In particular, the analysis about the effects of continual learning (Section 5.6) may not generalize to other types of function approximation that are not as prone to instability or divergence if over-trained. While evolutionary methods could in principle be combined with any kind of function approximation, in practice it is likely to work well only with very concise representations. Methods like CMACs, which use many more weights, would result in very large genomes and hence be difficult for evolutionary computation to optimize. However, since such methods methods become impractical as the number of state features and actions grow, concise methods like neural networks may become increasingly important in harder domains. If so, evolutionary function approximation could be an important tool for automatically optimizing their representations.

7. Related Work

A broad range of previous research is related in terms of both methods and goals to the techniques presented in this paper. This section highlights some of that research and contrasts it with this work.

7.1 Optimizing Representations for TD Methods

A major challenge of using TD methods is finding good representations for function approximators. This paper addresses that problem by coupling TD methods with evolutionary techniques like NEAT that are proven representation optimizers. However, many other approaches are also possible.

One strategy is to train the function approximator using supervised methods that also optimize representations. For example, Rivest and Precup (Rivest and Precup, 2003) train cascade-correlation networks as TD function approximators. Cascade-correlation networks are similar to NEAT in that they grow internal topologies for neural networks. However, instead of using evolutionary computation to find such topologies, they rely on the network's error on a given training set to compare alternative representations. The primary complication of Rivest and Precup's approach is that cascade-correlation networks, like many representation-optimizing supervised methods, need the training set to be both large and stable. TD methods do not naturally accommodate this requirement since they produce training examples only in sequence. Furthermore, those examples quickly become stale as the values upon which they were based are updated. Rivest and Precup address this problem using a novel caching system that in effect creates a hybrid value function consisting of a table and a neural network. While this approach delays the exploitation of the agent's experience, it nonetheless represents a promising way to marry the representation-optimizing capacity of cascade-correlation networks and other supervised algorithms with the power of TD methods.

Mahadevan (Mahadevan, 2005) suggests another strategy: using spectral analysis to derive basis functions for TD function approximators. His approach is similar to this work in that the agent is responsible for learning both the value function and its representation. It is different in that the representation is selected by analyzing the underlying structural properties of the state space, rather than evaluating potential representations in the domain.

A third approach is advanced by Sherstov and Stone (Sherstov and Stone, 2005): using the Bellman error generated by TD updates to assess the reliability of the function approximator in a given region of the state or action space. They use this metric to automatically adjust the breadth of generalization for a CMAC function approximator. An advantage of this approach is that feedback arrives immediately, since Bellman error can be computed after each update. A disadvantage is that the function approximator's representation is not selected based on its actual performance, which may correlate poorly with Bellman error.

There is also substantial research that focuses on optimizing the agent's state and action representations, rather than the value function representation. For example, Santamaria et al. (Santamaria et al., 1998) apply skewing functions to state-action pairs before feeding them as inputs to a function approximator. These skewing functions make the state-action spaces non-uniform and hence make it possible to give more resolution to the most critical regions. Using various skewing functions, they demonstrate improvement in the performance of TD learners. However, they do not offer any automatic way of determining how a given space should be skewed. Hence, a human designer still faces the burdensome task of manually choosing a representation, though in some domains using skewing functions may facilitate this process.

Smith (Smith, 2002) extends this work by introducing a method that uses self-organizing maps to automatically learn nonlinear skewing functions for the state-action spaces of TD agents. Self-organizing maps use unsupervised learning methods to create spatially organized internal representations of the inputs they receive. Hence, the system does not use any feedback on the performance of different skewing functions to determine which one is most appropriate. Instead it relies on the heuristic assumption that more resolution should be given to regions of the space that are more frequently visited. While this is an intuitive and reasonable heuristic, it does not hold in general. For example, a reinforcement learning agent designed to respond to rare emergencies may spend most of its life in safe states where its actions have little consequence and only occasionally experience crisis states where its choices are critical. Smith's heuristic would incorrectly devote most of its resolution to representing the value function of the unimportant but frequently visited states. Evolutionary function approximation avoids this problem because it evaluates competing representations by testing them in the actual task. It explicitly favors those representations that result in higher performance, regardless of whether they obey a given heuristic.

McCallum (McCallum, 1995) also presents a method for optimizing an agent's state representation. His approach automatically learns tree-structured short-term memories that allow reinforcement learning agents to prevent the state aliasing that results from hidden state.

7.2 Combining Evolutionary Computation with Other Learning Methods

Because of the potential performance gains offered by the Baldwin Effect, many researchers have developed methods that combine evolutionary computation with other learning methods that act within an individual's lifetime. Some of this work is applied to supervised problems, in which evolutionary computation can be coupled with any supervised learning technique such as backpropagation in a straightforward manner. For example, Boers et al. (Boers et al., 1995) introduce a neuroevolution technique that, like NEAT, tries to discover appropriate topologies. They combine this method with backpropagation and apply the result to a simple supervised learning problem. Also, Giraud-Carrier (Giraud-Carrier, 2000) uses a genetic algorithm to tune the parameters of RBF networks, which he applies to a supervised classification problem.

Inducing the Baldwin Effect on reinforcement learning problems is more challenging, since they do not automatically provide the target values necessary for supervised learning. The algorithms presented in this paper use TD methods to estimate those targets, though researchers have tried many other approaches. McQuestion and Miikkulainen (McQuesten and Miikkulainen, 1997) present a neuroevolutionary technique that relies on each individual's parents to supply targets and uses backpropagation to train towards those targets. Stanley et al. (Stanley et al., 2003) avoid the problem of generating targets by using Hebbian rules, an unsupervised technique, to change a neural network during its fitness evaluation. The network's changes are not directed by any error signal but they allow the network to retain a memory of previously experienced input sequences. Hence their approach is an alternative to recurrent neural networks. Downing (Downing, 2001) combines genetic programming with Q-learning using a simple tabular representation; genetic programming automatically learns how to discretize the state space.

Nolfi et al. (Nolfi et al., 1994) present a neuroevolutionary system that adds extra outputs to the network that are designed to predict what inputs will be presented next. When those inputs actually arrive, they serve as targets for backpropagation, which adjusts the network's weights starting from the added outputs. This technique allows a network to be adjusted during its lifetime using supervised methods but relies on the assumption that forcing it to learn to predict future inputs will help it select appropriate values for the remaining outputs, which actually control the agent's behavior. Another significant restriction is that the weights connecting hidden nodes to the action outputs cannot be adjusted at all during each fitness evaluation.

Ackley and Littman (Ackley and Littman, 1991) combine neuroevolution with reinforcement learning in an artificial life context. Evolutionary computation optimizes the initial weights of an "action network" that controls an agent in a foraging scenario. The weights of the network are updated during each individual's lifetime using a reinforcement learning algorithm called CRBP on the basis of a feedback signal that is also optimized with neuroevolution. Hence, their approach is similar to the one described in this paper, though the neuroevolution technique they employ does not optimize network topologies and CRBP does not learn a value function.

XCS Butz and Wilson (2002), based on learning classifier systems Lanzi et al. (2000), combine evolutionary computation and reinforcement learning in a different way. Each member of the population, instead of representing a complete policy, represents just a single classifier, which specifies the action the agent should take for some subset of the state space. Hence, the population as a whole represents a single evolving policy. Classifiers are selected for reproduction based on the accuracy of their value estimates and speciation is used to ensure the state space is properly covered.

Other combinations of evolutionary computation with other learning methods include Arita and Suzuki (Arita and Suzuki, 2000), who study iterated prisoner's dilemma; French and Messinger (French and Messinger, 1994) and Sasaki and Tokoro (Sasaki and Tokoro, 1999), who use artificial life domains; and Niv et al. (Niv et al., 2002) in a foraging bees domain.

Another important related method is VAPS Baird and Moore (1999). While it does not use evolutionary computation, it does combine TD methods with policy search methods. It provides a unified approach to reinforcement learning that uses gradient descent to try to simultaneously maximize reward and minimize error on Bellman residuals. A single parameter determines the relative weight of these goals. Because it integrates policy search and TD methods, VAPS is in much the same spirit as evolutionary function approximation. However, the resulting methods are quite different. While VAPS provides several impressive convergence guarantees, it does not address the question of how to represent the value function.

Other researchers have also sought to combine TD and policy search methods. For example, Sutton et al. (Sutton et al., 2000) use policy gradient methods to search policy space but rely on TD methods to obtain an unbiased estimate of the gradient. Similarly, in actor-critic methods Konda and Tsitsiklis (1999), the actor optimizes a parameterized policy by following a gradient informed by the critic's estimate of the value function. Like VAPS, these methods do not learn a representation for the value function.

7.3 Variable Evaluations in Evolutionary Computation

Because it allows members of the same population to receive different numbers of evaluations, the approach to on-line evolutionary computation presented here is similar to previous research about optimizing noisy fitness functions. For example, Stagge (Stagge, 1998) introduces mechanisms for deciding which individuals need more evaluations for the special case where the noise is Gaussian. Beielstein and Markon (Beielstein and Markon, 2002) 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 exploration/exploitation tradeoff, which has been studied extensively in the context of TD methods Watkins (1989); Sutton and Barto (1998) and multiarmed bandit problems Bellman (1956); Macready and Wolpert (1998); Auer et al. (2002). The selection mechanisms we employ in our system are well-established though, to our knowledge, their application to evolutionary computation is novel.

8. Future Work

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

Using Different Policy Search Methods This paper focuses on using evolutionary methods to automate the search for good function approximator representations. However, many other forms of policy search such as PEGASUS Ng and Jordan (2000) and policy gradient methods Sutton et al. (2000); Kohl and Stone (2004) 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 paper. In the future, we plan to test some of these alternative combinations.

Reducing Sample Complexity As mentioned in Section 6, 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 Lin (1992). 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.

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 we expect them to retain that capability when combined. In fact, we 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 paper 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.

Using Steady-State Evolutionary Computation The NEAT algorithm used in this paper 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 Fogarty (1989). 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 Stanley et al. (2005) so this extension is quite feasible.

9. Conclusion

Reinforcement learning is an appealing and empirically successful approach to finding effective control policies in large probabilistic domains. However, it requires a good deal of expert knowledge to put into practice, due in large part to the need for manually defining function approximator representations. This paper offers hope that machine learning methods can be used to discover those representations automatically, thus broadening the practical applicability of reinforcement learning.

This paper makes three main contributions. First, it introduces evolutionary function approximation, which automatically discovers effective representations for TD function approximators. Second, it introduces on-line evolutionary computation, which employs selection mechanisms borrowed from TD methods to improve the on-line performance of evolutionary computation. Third, it provides a detailed empirical study of these methods in the mountain car and server job scheduling domains.

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. Combined, our novel algorithms offer a promising and general approach to reinforcement learning in large probabilistic domains.

Acknowledgments

Thanks to Richard Sutton, Michael Littman, Gerry Tesauro, and Manuela Veloso for helpful discussions and ideas. Thanks to Risto Miikkulainen, Nick Jong, Bikram Banerjee, Shivaram Kalyanakrishnan, and the anonymous reviewers for constructive comments about earlier versions of this work. This research was supported in part by NSF CAREER award IIS-0237699 and an IBM faculty award.

Appendix A. Statistical Significance

To assess the statistical significance of the results presented in Section 5, we performed a series of Student's t-tests on each pair of methods in each domain. For each pair, we performed a t-test after every 100,000 episodes. Tables 1 and 2 summarize the results of these tests for the mountain car and server job scheduling domains, respectively. In each table, the values in each cell indicate the range of episodes for which performance differences were significant with 95% confidence.

Episodes (x1000)	Q-Learning	Off-Line NEAT	ϵ-Greedy NEAT	Softmax NEAT	Off-Line NEAT+Q	Softmax NEAT+Q	Lamarckian NEAT+Q
Q-Learning							
Off-Line NEAT	300 to 1000						
ϵ-Greedy NEAT	200 to 1000	200 to 1000					
Softmax NEAT	200 to 1000	200 to 1000	200 to 1000				
Off-Line NEAT+Q	200 to 1000	200 to 500	200 to 1000	200 to 1000			
Softmax NEAT+Q	100 to 1000	200 to 1000	200 to 1000	900 to 1000	200 to 1000		
Lamarkian NEAT+Q	200 to 1000	200 to 1000	200 to 1000	200 to 1000	200 to 1000	100 to 1000	

Table 1: A summary of the statistical significance of differences in average performance between each pair of methods in mountain car (see Figures 4, 6, 7 & 10). Values in each cell indicate the range of episodes for which differences were significant with 95% confidence.

Episodes (x1000)	Q-Learning	Off-Line NEAT	ϵ-Greedy NEAT	Softmax NEAT	Off-Line NEAT+Q	Softmax NEAT+Q	Lamarckian NEAT+Q
Q-Learning							
Off-Line NEAT	300 to 1000						
ϵ-Greedy NEAT	200 to 1000	200 to 1000					
Softmax NEAT	200 to 1000	200 to 1000	not significant throughout				
Off-Line NEAT+Q	300 to 1000	300 to 500	100 to 1000	200 to 1000			
Softmax NEAT+Q	200 to 1000	200 to 1000	400 to 1000	200 to 1000	200 to 1000		
Lamarckian NEAT+Q	300 to 1000	300 to 1000	100 to 1000	100 to 1000	700 to 1000	200 to 1000	

Table 2: A summary of the statistical significance of differences in average performance between each pair of methods in server job scheduling (see Figures 4, 6, 7 & 10). Values in each cell indicate the range of episodes for which differences were significant with 95% confidence.

Appendix B. NEAT Parameters

Table 3 details the NEAT parameters used in our experiments. Stanley and Miikkulainen (Stanley and Miikkulainen, 2002) describe the semantics of these parameters in detail.

Parameter	Value	Parameter	Value	Parameter	Value
weight-mut-power	0.5	recur-prop	0.0	disjoint-coeff (c_1)	1.0
excess-coeff (c_2)	1.0	mutdiff-coeff (c_3)	2.0	compat-threshold	3.0
age-significance	1.0	survival-thresh	0.2	mutate-only-prob	0.25
mutate-link-weights-prob	0.9	mutate-add-node-prob (m_n)	0.02	mutate-add-link-prob (m_l)	0.1
interspecies-mate-rate	0.01	mate-multipoint-prob	0.6	mate-multipoint-avg-prob	0.4
mate-singlepoint-prob	0.0	mate-only-prob	0.2	recur-only-prob	0.0
pop-size (p)	100	dropoff-age	100	newlink-tries	50
babies-stolen	0	num-compat-mod	0.3	num-species-target	6

Table 3: The NEAT parameters used in our experiments.

References

David Ackley and Michael Littman. Interactions between learning and evolution. *Artificial Life II, SFI Studies in the Sciences of Complexity*, 10:487–509, 1991.

T. Arita and R. Suzuki. Interactions between learning and evolution: The outstanding strategy generated by the Baldwin Effect. *Artificial Life*, 7:196–205, 2000.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

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

Leemon Baird and Andrew Moore. Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems 11*. MIT Press, 1999.

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

Thomas Beielstein and Sandor Markon. Threshold selection, hypothesis tests and DOE methods. In *2002 Congresss on Evolutionary Computation*, pages 777–782, 2002.

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

E.J.W. Boers, M.V. Borst, and I.G. Sprinkhuizen-Kuyper. Evolving Artificial Neural Networks using the “Baldwin Effect”. In *Artificial Neural Nets and Genetic Algorithms, Proceedings of the International Conference in Ales, France*, 1995.

Justin A. Boyan and Michael L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 671–678. Morgan Kaufmann Publishers, Inc., 1994.

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

Martin V. Butz and Stewart W. Wilson. An algorithmic description of XCS. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 6(3-4):144–153, 2002.

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

K. Mathias D. Whitley, S. Gordon. Lamarckian evolution, the Baldwin effect and function optimization. In *Parallel Problem Solving from Nature - PPSN III*, pages 6–15, 1994.

Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. Adverserial classification. In *Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 99–108, 2004.

Keith L. Downing. Reinforced genetic programming. *Genetic Programming and Evolvable Machines*, 2(3):259–288, 2001.

L. Ertoz, A. Lazarevic, E. Eilerston, A. Lazarevic, P. Tan, P. Dokas, V. Kumar, and J. Srivastava. *The MINDS - Minnesota Intrustion Detection System*, chapter 3. MIT Press, 2004.

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.

Robert French and Adam Messinger. Genes, phenes and the Baldwin effect: Learning and evolution in a simulated population. *Artificial Life*, 4:277–282, 1994.

Christophe Giraud-Carrier. Unifying learning with evolution through Baldwinian evolution and Lamarckism: A case study. In *Proceedings of the Symposium on Computational Intelligence and Learning (CoIL-2000)*, pages 36–41, 2000.

David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1989.

David 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.

Faustino Gomez, Doug Burger, and Risto Miikkulainen. A neuroevolution method for dynamic resource allocation on a chip multiprocessor. In *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, pages 2355–2361, 2001.

Frederic Gruau and Darrell Whitley. Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. *Evolutionary Computation*, 1:213–233, 1993.

Frederic Gruau, Darrell Whitley, and Larry 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.

Geoffrey E. Hinton and Steven J. Nowlan. How learning can guide evolution. *Complex Systems*, 1:495–502, 1987.

John 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.

Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems 11*, pages 1008–1014, 1999.

R. Matthew Kretchmar and Charles W. Anderson. Comparison of CMACs and radial basis functions for local function approximators in reinforcement learning. In *International Conference on Neural Networks*, 1997.

Michail G. Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4(2003):1107–1149, 2003.

Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart Wilson. *Learning classifier systems from foundations to applications*. Springer, 2000.

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

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association of Computing Machinery*, 20(1): 46–61, January 1973.

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

Sridhar Mahadevan. Samuel meets Amarel: Automating value function approximation using global state space analysis. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, 2005.

S. Mannor, R. Rubenstein, and Y. Gat. The cross-entropy method for fast policy search. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 512–519, 2003.

Andrew R. McCallum. Instance-based utile distinctions for reinforcement learning. In *Proceedings of the Twelfth International Machine Learning Conference*, 1995.

Amy McGovern, Eliot Moss, and Andrew G. Barto. Building a block scheduler using reinforcement learning and rollouts. *Machine Learning*, 49(2-3):141–160, 2002.

Paul McQuesten and Risto Miikkulainen. Culling and teaching in neuro-evolution. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 760–767, 1997.

David McWherter, Bianca Schroeder, Natassa Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the Twentieth International Conference on Data Engineering*, 2004.

Andrew Y. Ng and Michael 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.

Y. Niv, D. Joel, I. Meilijson, and E. Ruppin. Evolution of reinforcement learning in foraging bees: A simple explanation for risk averse behavior. *Neurocomputing*, 44(1):951–956, 2002.

Stefano Nolfi, Jeffery L. Elman, and Domenico Parisi. Learning and evolution in neural networks. *Adaptive Behavior*, 2:5–28, 1994.

F. B. Pereira and E. Costa. Understanding the role of learning in the evolution of busy beaver: A comparison between the Baldwin Effect and a Lamarckian strategy. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

Larry D. Pyeatt and Adele E. Howe. Decision tree function approximation in reinforcement learning. In *Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models*, pages 70–77, 2001.

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

Martin Reidmiller. Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. In *Proceedings of the Sixteenth European Conference on Machine Learning*, pages 317–328, 2005.

F. Rivest and D. Precup. Combining TD-learning with cascade-correlation networks. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 632–639. AAAI Press, 2003.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, pages 318–362. 1986.

J. Santamaria, R. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2), 1998.

Takahiro Sasaki and Mario Tokoro. Evolving learnable neural networks under changing environments with various rates of inheritance of acquired characters: Comparison between Darwinian and Lamarckian evolution. *Artificial Life*, 5(3):203–223, 1999.

Alexander A. Sherstov and Peter Stone. Function approximation via tile coding: Automating parameter choice. In J.-D. Zucker and I. Saitta, editors, *SARA 2005*, volume 3607 of *Lecture Notes in Artificial Intelligence*, pages 194–205. Springer Verlag, Berlin, 2005.

William D. Smart and Leslie Pack Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeeth International Conference on Machine Learning*, pages 903–910, 2000.

Andrew James Smith. Applications of the self-organizing map to reinforcement learning. *Journal of Neural Networks*, 15:1107–1124, 2002.

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

Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving adaptive neural networks with and without adaptive synapses. In *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, volume 4, pages 2557–2564, 2003.

Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving neural network agents in the NERO video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, 2005.

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

Kenneth O. Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004a.

Kenneth O. Stanley and Risto Miikkulainen. Evolving a roving eye for go. In *Proceedinngs of the Genetic and Evolutionary Computation Conference*, 2004b.

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

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

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

R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

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

Jan A. van Mieghem. Dynamic scheduling with convex delay costs: The generalized $c\mu$ rule. *The Annals of Applied Probability*, 5(3):809–833, August 1995.

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.

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

Shimon Whiteson and Peter Stone. Adaptive job routing and scheduling. *Engineering Applications of Artificial Intelligence*, 17(7):855–869, 2004. Corrected version.

Jonathan Wildstrom, Peter Stone, Emmett Witchel, Raymond J. Mooney, and Mike Dahlin. Towards self-configuring hardware for distributed computer systems. In *The Second International Conference on Autonomic Computing*, pages 241–249, June 2005.

K. Yamasaki and M. Sekiguchi. Clear explanation of different adaptive behaviors between Darwinian population and Lamarckian population in changing environment. In *Proceedings of the Fifth International Symposium on Artificial Life and Robotics*, volume 1, pages 120–123, 2000.

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

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 1995 Joint Conference on Artificial Intelligence*, pages 1114–1120, 1995.

Monte Zweben and Mark Fox, editors. *Intelligent Scheduling*. Morgan Kaufmann, 1998.