Evolutionary Function Approximation for Reinforcement Learning

Shimon Whiteson and Peter Stone
Department of Computer Sciences
University of Texas at Austin
1 University Station, C0500
Austin, TX 78712-0233
{shimon,pstone}@cs.utexas.edu,
http://www.cs.utexas.edu/~{shimon,pstone}

Abstract. Temporal difference methods are theoretically grounded and empirically effective methods for addressing sequential decision making problems with delayed rewards. Most problems of real-world interest require coupling TD methods with a function approximator to represent the value function. However, using function approximators requires manually making crucial representational decisions. This paper introduces *evolutionary function approximation*, a novel approach to automatically selecting function approximator representations that enable efficient individual learning. Our 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, and Q-learning, a popular temporal difference method. The resulting NEAT+Q algorithm automatically learns effective representations for neural network function approximators. Empirical results in a server job scheduling task demonstrate that NEAT+Q can significantly improve the performance of TD methods.

1 Introduction

In many machine learning problems, an agent must learn a *policy* for selecting actions based on its *state*, which consists of its current knowledge about the world. *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 feedback 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 *temporal difference methods* (TD) [1], which use dynamic programming and statistical sampling to estimate the long-term value of taking each possible action in each possible state. Once this value function has been learned, an effective policy can be trivially derived. Hence, TD methods enable an individual agent to learn during its "lifetime" i.e. its experience interacting with the environment.

For small problems, the value function can be represented as a table. However, most problems of real-world interest 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 [1]. However, using function approximators requires making crucial representational decisions (e.g. the number of hidden units and initial weights of a neural network). Poor design choices can result in estimates that diverge from the optimal value function [2] and agents that perform poorly. These crucial decisions are typically made manually, based only on the designer's intuition.

This paper introduces *evolutionary function approximation*, a population-based approach to automatically selecting function approximator representations that enable efficient individual learning. Our method *evolves* individuals that are better able to *learn*. This biologically intuitive combination has been applied to computational systems in the past [3–5] but never, to our knowledge, to aid the discovery of good function approximators.

Our approach requires only 1) an evolutionary algorithm capable of learning representations from a class of functions (e.g. neural networks) and 2) a TD method that uses elements of that class for function approximation. In this paper, we use NeuroEvolution of Augmenting Topologies (NEAT) [6] in conjunction with Q-learning, a popular TD method that has been successfully coupled with neural network function approximators [7, 8]. The resulting algorithm, 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.

This paper introduces and fully specifies evolutionary function approximation and NEAT+Q, and presents a detailed empirical analysis from the domain of server job scheduling, a challenging reinforcement learning task from the burgeoning field of *autonomic computing* [9]. Our experiments demonstrate that, in at least one domain, evolutionary function approximation can significantly improve the performance of TD methods.

2 Background

This section reviews the two algorithms that form the building blocks of our implementation of evolutionary function approximation: Q-learning and NEAT.

2.1 Q-Learning

There are several different TD methods currently in use, including Q-learning, Sarsa, and TD(λ) [1]. The experiments presented in this paper use Q-learning because it is a canonical method and has achieved notable empirical success when combined with neural network function approximators [7, 8]. We present it as a representative method but do not claim it is better than other TD approaches. In principle, evolutionary function approximation can be used with any of them.

Like other TD methods, Q-learning attempts to learn a value function 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':

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

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. S is the set of states, A is the set of actions, and e is the number of episodes for which the algorithm runs. 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. After each state transition (line 8), the weights of the neural network are adjusted using backpropagation (lines 9–11) such that its output better matches the current value estimate for the state-action pair: $r + \gamma \max_{a'} Q(s', a')$. This implementation uses ϵ-greedy exploration [1] to ensure the agent occasionally tests alternatives to its current policy (lines 6–7).

Algorithm 1 Q-LEARN$(S, A, \alpha, \gamma, \epsilon, e)$

```
 1: N ← INIT-NET(S, A)
 2: for i ← 1 to e do
 3:    s ← INIT-STATE(S)
 4:    Q[] ← EVAL-NET(N, s)
 5:    while ¬TERMINAL-STATE?(s) do
 6:       with-prob(ε) a ← RANDOM(A)
 7:       else a ← argmax_{i←0}^{i<|Q|} Q[i]
 8:       r, s' ← TAKE-ACTION(a)
 9:       Q[] ← EVAL-NET(N, s')
10:       v_{a'} ← max_{i←0}^{i<|Q|} Q[i]
11:       BACKPROP(N, s, a, r + γ v_{a'}, α)
12:       s ← s'
```

Function approximators like neural networks allow Q-learning to be applied to problems with more state-action pairs than can feasibly be represented in a table. However, getting them to work requires manually selecting the network's topology and initial weights. The difficulty of doing so is one of the chief motivations for evolutionary function approximation, which relies on methods like NEAT to automatically learn effective topologies. The next section describes the NEAT algorithm.

2.2 NEAT

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 pole balancing [6], game playing [10], and robot control [11], and because of its ability to automatically learn network topologies.

In a typical neuroevolutionary system [12], 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.

Since NEAT is a general purpose optimization technique, it can be applied to a wide variety of problems. Section 3 below describes how we use NEAT to learn the topology and initial weights of Q-learning's function approximators. Here, we describe how NEAT can be used, without the aid of Q-learning, to tackle reinforcement learning problems, an approach that serves as one baseline of comparison in Section 4. For this method, an example of *policy search* reinforcement learning, NEAT does not attempt to learn a value function. Instead, it learns a policy directly by training *action selectors* that directly map states to the action the agent should take in that state.

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

```
 1: P ← INIT-POPULATION(S, A, p)
 2: for i ← 0 to g do
 3:    for j ← 0 to e do
 4:       N ← RANDOM(P)
 5:       s ← INIT-STATE(S)
 6:       Q[] ← EVAL-NET(N, s)
 7:       while ¬TERMINAL-STATE?(s) do
 8:          a ← argmax_{i=0}^{i<|Q|} Q[i]
 9:          r, s' ← TAKE-ACTION(a)
10:          Q[] ← EVAL-NET(N, s')
11:          N.fitness ← N.fitness + r
12:          s ← s'
13:       N.episodes ← N.episodes + 1
14:    P ← BREED-NEW-POPULATION(P)
```

Algorithm 2 contains a high-level description of the NEAT algorithm applied to an episodic reinforcement learning problem. In this algorithm, p is the size of the population, g is the number of generations evolution lasts, and e is the number of episodes performed in each generation. In each episode, a member of the population is randomly selected for evaluation (line 4). During each step, the agent takes whatever action corresponds to the output with the highest activation (lines 8–9). NEAT maintains a running total of the reward accrued by the network during its evaluation (line 11). NEAT differs from other neuroevolution techniques in how it creates a new generation of networks (line 14) based on the average fitness ($N.fitness/N.episodes$) of each member of the previous generation. The remainder of this section gives a brief overview of this process. More details can be found in [6].

Unlike other systems that evolve network topologies and weights [12], NEAT begins with a uniform population of simple networks with no hidden nodes and inputs connected directly to outputs. Two special mutation operators introduce new structure incrementally. Figure 1 depicts these operators, which add hidden nodes and links to the network. Only those structural mutations that improve performance tend to survive; in this way, NEAT searches through a minimal number of weight dimensions and finds the appropriate complexity level for the problem.

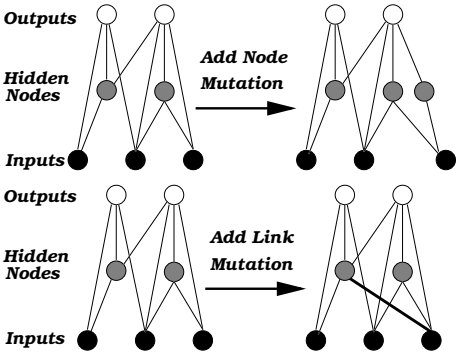


Fig. 1. Examples of NEAT's mutation operators for adding structure to networks. At top, a hidden node is added by splitting a link in two. At bottom, a link, shown with a thicker black line, is added to connect two nodes.

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

3 Method

This section introduces evolutionary function approximation, a new approach that enables the automatic discovery of function approximator representations. When evolutionary systems 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 the evolutionary system is directed to learn value functions instead, then those value functions can be updated, using TD methods, during each fitness evaluation.

In addition to automating the search for effective representations, evolutionary function approximation makes it possible to exploit the Baldwin Effect, a phenomenon whereby populations whose individuals learn during their lifetime adapt more quickly than populations whose individuals remain static [13]. This effect, which has been demonstrated in evolutionary computation [3, 4], results in faster evolution because an individual does not have to be exactly right at birth; it need only be in the right neighborhood and learning can adjust it accordingly. Hence, the Baldwin Effect results from a synergy between learning *across* fitness evaluations and learning *within* fitness evaluations. Due to this synergy, we expect evolutionary function approximation, in addition to improv-

ing the performance of TD methods, to outperform evolutionary methods that train fixed action selectors.

The remainder of this section presents NEAT+Q, the particular implementation of evolutionary function approximation which is empirically evaluated in this paper.

3.1 NEAT+Q

All that is required to make NEAT learn 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 value 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 that, each time the agent takes an action, the network is backpropagated towards Q-learning targets (lines 11–13) and ϵ-greedy exploration occurs just as in Algorithm 1 (lines 8-9). If α and ϵ are set to zero, this method degenerates to regular NEAT.

Algorithm 3 NEAT+Q$(S, A, p, g, e, \alpha, \gamma, \epsilon)$

```
 1: P ← INIT-POPULATION(S, A, p)
 2: for i ← 0 to g do
 3:    for j ← 0 to e do
 4:       N ← RANDOM(P)
 5:       s ← INIT-STATE(S)
 6:       Q[] ← EVAL-NET(N, s)
 7:       while ¬TERMINAL-STATE?(s) do
 8:          with-prob(ϵ) a ← RANDOM(A)
 9:          else a ← argmax_{i←0}^{i<|Q|} Q[i]
10:          r, s' ← TAKE-ACTION(a)
11:          Q[] ← EVAL-NET(N, s')
12:          v_{a'} ← max_{i←0}^{i<|Q|} Q[i]
13:          BACKPROP(N, s, a, r + γ v_{a'}, α)
14:          N.fitness ← N.fitness + r
15:          s ← s'
16:       N.episodes ← N.episodes + 1
17:    P ← BREED-NEW-POPULATION(P)
```

Hence, 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. At the end of a generation, those changes are *not* written back into the networks' genomes. This *Darwinian* approach contrasts with a *Lamarckian* one in which an individual's learning affects its genome and is inherited by its offspring. Comparing our approach to a Lamarckian one is an important part of our ongoing research.

4 Experiments

In this section, we apply NEAT+Q to a challenging task motivated by a real-world application and compare it to Q-learning with several manually designed function approximator topologies. We test whether NEAT+Q can automatically find networks that perform as well as or better than the best networks resulting from a manual search. We also test whether NEAT+Q can outperform regular NEAT, as predicted by the Baldwin Effect.

As a test domain, we use the autonomic computing task of server job scheduling. The goal of autonomic computing [9] is to develop computer systems that automatically optimize their performance and diagnose and repair their own failures. In server job scheduling [14], a server, such as a website's application server or database, must determine in what order to process the jobs currently waiting in its queue. Its goal is to maximize the aggregate utility of all the jobs it processes. A *utility function* for each job type maps the job's completion time to the utility the derived by the user. The problem of server job scheduling becomes challenging when these utility functions are non-linear and/or the server must process multiple types of jobs. Since selecting a particular job for processing necessarily delays the completion of all other jobs in the queue, the scheduler must weigh difficult trade-offs to maximize aggregate utility.

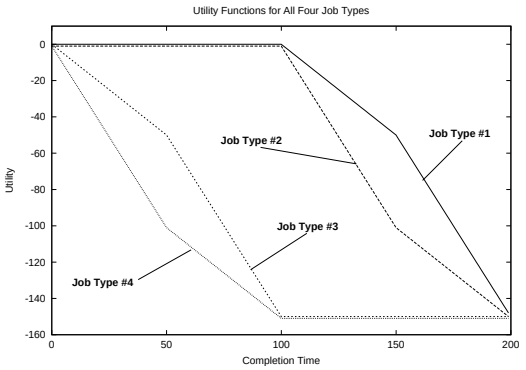


Fig. 2. The four utility functions

Our experiments were conducted in a Java-based simulator. During each timestep, the server removes one job from its queue and completes it. During the first 100 timesteps, a new job of a randomly selected type is added to the end of the queue. The simulation begins with 100 jobs preloaded into the server's

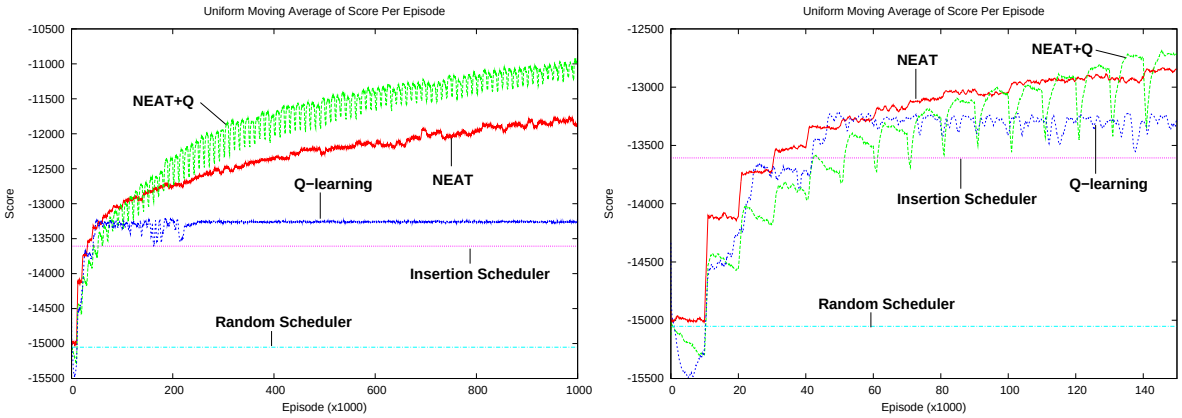
queue and ends when the queue becomes empty at timestep 200. For each job that completes, the scheduling agent receives an immediate reward determined by that job's utility function. Our experiments use four different job types, whose utility functions are shown in Figure 2. Users who create jobs of 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 of 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.

Using this setup, we conducted 10 runs in which NEAT attempts to learn a scheduler. In these runs, the population size p was 100, the number of generations g was 100, and the number of episodes per generation e was 10,000. Hence, each individual was evaluated for 100 episodes on average. Then we performed 10 runs using NEAT+Q with the same parameter settings. In addition, the learning rate α was 0.1, the discount factor γ was 0.95, and the exploration rate ϵ was 0.05. To test Q-learning alone, we selected three different network topologies: feed-forward networks with 0, 4, or 8 hidden nodes. For each topology, we tested three different values of α: 0.01, 0.05, and 0.1. For each value of α, we tested Q-learning with and without an annealing schedule. When the schedule was used, α decayed linearly until reaching zero after 250,000 episodes. The other parameters, γ and ϵ were set to 0.95 and 0.05 respectively, just as with NEAT+Q. Each of these 18 configurations were evaluated for 10 runs. In addition, we experimented informally with higher values of γ, slower linear annealing, and exponential annealing, though none performed as well as the results presented here.

Figure 3 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 10 runs. For the sake of clarity, only the highest performing configuration of Q-learning (4 hidden nodes and $\alpha = 0.01$) is included in the figure. The graph also shows

the performance of two heuristic schedulers, averaged over 10,000 episodes. The random scheduler, which is included to establish a baseline of comparison, randomly selects jobs for processing. The insertion scheduler [14] 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. Every time a new job is created, the insertion scheduler tries inserting it into each position in the queue, settling on whichever position it estimates will yield the highest aggregate utility.



(a) Performance over 1 million episodes of learning (b) Close−up of first 150,00 episodes

Fig. 3. A comparison of the performance of NEAT, Q-learning, and NEAT+Q in the server job scheduling domain.

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. Figure 3b, which shows a close-up of the early part of the same runs, reveals that these changes are actually harmful in early generations of NEAT+Q but, over time, begin to improve performance as NEAT+Q learns appropriate topologies and initial weights for Q-learning.

5 Discussion

Figure 3a demonstrates that all three learning methods outperform the heuristic schedulers, including one that performed best in a previous study in this domain [14]. The graph also shows that NEAT+Q achieves the highest performance of all the methods tested. Indeed, a Student's t-test confirms, with

95% confidence, that the difference between NEAT+Q and the other methods is statistically significant after 350,000 episodes.

For the particular problem we tested and network configurations we tried, NEAT+Q substantially improves performance over Q-learning with manually designed networks. Of course, 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 the best network discovered by NEAT+Q and the learning rate is annealed aggressively, Q-learning matches NEAT+Q's performance without directly using evolution. However, it is unlikely that this successful topology, which contains irregular and partially connected hidden layers, would be discovered through a manual search process, no matter how extensive.

Figure 3b reveals that even in early episodes NEAT+Q learns at approximately the same rate as Q-learning. After about 100,000 episodes, Q-learning plateaus while NEAT+Q continues to improve.[1] While it may be possible to improve Q-learning's peak performance with further manual tuning, NEAT+Q achieves that improvement via additional computation.

NEAT+Q also substantially outperforms regular NEAT, which highlights the value of temporal difference methods on challenging reinforcement learning problems. Even when NEAT is employed to learn 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.

Note that the advantage of NEAT+Q over NEAT is not directly explained just by the learning that occurs via backpropagation within each generation. By generation 20, 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.

6 Related Work

Some previous research has aimed, as does this paper, at automatically learning representations for function approximators. For example, Smith [15] uses self-organizing maps to automatically learn non-linear skewing functions for state-action spaces. However, unlike our work, this approach relies on the heuristic assumption that more resolution should be given to regions of the space that are more frequently visited. Another approach is to train function approximators using supervised methods that also learn representations, as Rivest and

[1] After 250,000 episodes, Q-learning's learning rate has annealed to zero and no additional learning is possible. However, in our experiments, running Q-learning with slower annealing or no annealing at all consistently led to inferior and unstable performance.

Precup [16] do with cascade-correlation networks. The primary complication is that cascade-correlation networks, like many representation-learning supervised methods, require a large and stable training set, which TD methods do not provide. Rivest and Precup address this problem using a novel caching system. While this approach delays the exploitation of the agent's experience, it nonetheless represents a promising way to marry representation-learning supervised algorithms with TD methods.

Furthermore, many researchers have tried to exploit the Baldwin Effect by combining evolutionary computation with other learning methods. Most of these results, such as [4], are in supervised problems where it is straightforward to synthesize evolution with standard techniques. In reinforcement learning problems, it is less clear how to generate the target values necessary for learning during an individual's lifetime. While this paper uses value estimates generated by TD methods as targets, Nolfi et al. [5] train state predictors and use the actual states as targets, while Stanley et al. [17] use unsupervised methods that do not require targets [17] and Ackley and Littman [3] use reinforcement learning techniques that do not learn value functions.

Finally, VAPS [18] is an algorithm that, like NEAT+Q, integrates TD and policy search methods. It yields impressive convergence guarantees but does not address the problem of learning appropriate representations for value functions or policies.

7 Ongoing and Future Work

Though space restrictions prevent their full presentation here, we have tested Sarsa, NEAT+Sarsa, and a Lamarckian version of NEAT+Q in the server scheduling domain. Sarsa and NEAT+Sarsa perform similarly to, though not as well as, their Q-learning counterparts. Surprisingly, Lamarckian NEAT+Q performs dramatically worse than the Darwinian version presented here. More investigation is necessary to determine why, though we suspect it is related to the difficulty of finding network weights that remain stable over the long term.

Future research will focus on implementing evolutionary function approximation with different constituent learning methods. Many other TD methods could be used in place of Q-learning and Sarsa. Also, we hope to employ evolutionary systems other than NEAT, as doing so could enable the automatic configuration of function approximators other than neural networks. Furthermore, we hope to evaluate evolutionary function approximation on additional domains to further establish the scope of its efficacy.

8 Conclusion

The primary contribution of this paper is evolutionary function approximation, which uses evolution to automate the search for effective representations for TD function approximators. A particular implementation called NEAT+Q automatically discovers effective topologies for neural network function approximators for Q-learning. Empirical results in server job scheduling, an autonomic computing domain, demonstrate that NEAT+Q performs significantly better than both

NEAT and Q-learning, even after an extensive manual search for effective function approximators. Hence, evolutionary function approximation can eliminate the need for manually specifying function approximator representations.

Acknowledgments

Thanks to Richard Sutton for helpful discussions and insights. This research was supported in part by NSF CAREER award IIS-0237699 and an IBM faculty award.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
2. Baird, L.: Residual algorithms: Reinforcement learning with function approximation. In: Proceedings of the Twelfth International Conference on Machine Learning, Morgan Kaufmann (1995) 30–37
3. Ackley, D., Littman, M.: Interactions between learning and evolution. Artificial Life II, SFI Studies in the Sciences of Complexity **10** (1991) 487–509
4. Boers, E., Borst, M., Sprinkhuizen-Kuyper, I.: Evolving Artificial Neural Networks using the "Baldwin Effect". Technical Report TR 95-14 (1995)
5. Nolfi, S., Elman, J.L., Parisi, D.: Learning and evolution in neural networks. Adaptive Behavior **2** (1994) 5–28
6. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10** (2002) 99–127
7. Crites, R.H., Barto, A.G.: Elevator group control using multiple reinforcement learning agents. Machine Learning **33** (1998) 235–262
8. Tesauro, G.: TD-Gammon, a self-teaching backgammon program, achieves master-level play. Neural Computation **6** (1994) 215–219
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36(1)** (2003) 41–50
10. Stanley, K.O., Miikkulainen, R.: Evolving a roving eye for go. In: Proceedinngs of the Genetic and Evolutionary Computation Conference. (2004)
11. Stanley, K.O., Miikkulainen, R.: Competitive coevolution through evolutionary complexification. Journal of Artificial Intelligence Research **21** (2004) In press.
12. Yao, X.: Evolving artificial neural networks. Proceedings of the IEEE **87** (1999) 1423–1447
13. Baldwin, J.M.: A new factor in evolution. The American Naturalist **30** (1896) 441–451
14. Whiteson, S., Stone, P.: Adaptive job routing and scheduling. Engineering Applications of Artificial Intelligence **17(7)** (2004) 855–869
15. Smith, A.J.: Applications of the self-organizing map to reinforcement learning. Journal of Neural Networks **15** (2002) 1107–1124
16. Rivest, F., Precup, D.: Combining td-learning with cascade-correlation networks. In: Proceedings of the Twentieth International Conference on Machine Learning, AAAI Press (2003) 632–639
17. Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Evolving adaptive neural networks with and without adaptive synapses. In: Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003). Volume 4. (2003) 2557–2564
18. Baird, L., Moore, A.: Gradient descent for general reinforcement learning. In: Advances in Neural Information Processing Systems 11, MIT Press (1999)