In *Proceedings of the Twenty-First National Conference on Artificial Intelligence* (AAAI 06),
pp. 518-523, Boston, MA, July 2006.

Sample-Efficient 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

Abstract

Reinforcement learning problems are commonly tackled with temporal difference methods, which attempt to estimate the agent's optimal value function. In most real-world problems, learning this value function requires a function approximator, which maps state-action pairs to values via a concise, parameterized function. In practice, the success of function approximators depends on the ability of the human designer to select an appropriate representation for the value function. A recently developed approach called evolutionary function approximation uses evolutionary computation to automate the search for effective representations. While this approach can substantially improve the performance of TD methods, it requires many sample episodes to do so. We present an enhancement to evolutionary function approximation that makes it much more sample-efficient by exploiting the off-policy nature of certain TD methods. Empirical results in a server job scheduling domain demonstrate that the enhanced method can learn better policies than evolution or TD methods alone and can do so in many fewer episodes than standard evolutionary function approximation.

Introduction

Reinforcement learning problems are typically solved using *temporal difference* (TD) methods (Sutton & Barto 1998). These methods rely on *value functions*, which indicate, for a particular policy, the long-term expected value of a given state or state-action pair. For small problems, the value function can be stored in a table. However, real-world problems usually require *function approximators*, which map state-action pairs to values via concise, parameterized functions.

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 & Moore 1999; Lagoudakis & Parr 2003), achieving high performance in practice requires finding an appropriate representation for the function approximator. Nonetheless, representational choices are typically made manually, based only on the designer's intuition.

A recently developed approach called *evolutionary function approximation* (Whiteson & Stone 2006) uses evolutionary computation (Goldberg 1989) to automate the search for effective representations. In this approach, evolution automatically selects function approximator representations that enable efficient individual learning. Thus, it *evolves* agents that are better able to *learn*. Empirical results have demonstrated that evolutionary function approximation can significantly outperform both evolutionary and TD methods alone (Whiteson & Stone 2006).

Evolutionary function approximation suffers from one important disadvantage: high sample complexity. Each candidate representation in the population must be evaluated for many episodes before TD updates have a significant effect. High sample complexity is undesirable because sample episodes are typically the scarcest resource: each new episode may incur substantial real-world costs whereas additional memory and CPU cycles are relatively inexpensive.

In this paper, we present an enhancement to evolutionary function approximation designed to make it dramatically more sample-efficient. This enhancement relies on TD methods that are *off-policy*, i.e. that can estimate the optimal value function regardless of what policy the agent is following. By storing experience from the previous generation, sample-efficient evolutionary function approximation can train each new generation off-line using only computation time: no additional sample episodes are needed. The resulting function approximators can then be evaluated and selectively reproduced in many fewer episodes.

We present an implementation of this method called sample-efficient NEAT+Q, which uses NEAT (Stanley & Miikkulainen 2002), a neuroevolutionary algorithm, to select neural network function approximators for Q-learning (Watkins 1989), a popular off-policy TD method. Empirical results in an autonomic computing domain called server job scheduling demonstrate that sample-efficient NEAT+Q can learn better policies than NEAT or Q-learning alone and can do so in many fewer episodes than the original NEAT+Q approach.

*We wish to thank Matt Taylor, Nick Jong and the anonymous reviewers for helpful comments about earlier versions of this paper. This research was supported in part by NSF CAREER award IIS-0237699 and an IBM faculty award.

Background

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

Q-Learning

The experiments presented in this paper use Q-learning because it is a well-established, canonical TD method that has also enjoyed empirical success (Watkins 1989; Crites & Barto 1998). 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. Q-learning is an *off-policy* learning method, i.e. it can learn the optimal value function regardless of what policy the agent is following, so long as there is sufficient exploration. As explained in the Method section below, this feature of Q-learning is critical to making evolutionary function approximation sample-efficient.

In domains with large or continuous state spaces, the value function cannot be represented in a table. Instead, Q-learning is coupled with a function approximator that maps state-action pairs to values via a concise, parameterized function. Many different methods of function approximation have been used successfully, including CMACs, radial basis functions, and neural networks (Sutton & Barto 1998). In this paper, we use neural network function approximators because they have proven successful on difficult reinforcement learning tasks (Tesauro 1994; Crites & Barto 1998). The inputs to the network describe the agent's current state; the outputs, one for each action, represent the agent's current estimate of the value of the associated state-action pairs. The initial weights of the network are drawn from a Gaussian distribution with mean 0.0 and standard deviation σ. After each action, the weights of the neural network are adjusted using backpropagation (Rumelhart, Hinton, & Williams 1986) such that its output better matches the current value estimate for the state-action pair: $r + \gamma \max_{a'} Q(s', a')$.

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 success on difficult reinforcement learning tasks (Stanley & Miikkulainen 2002; Whiteson & Stone 2006). In addition, unlike many other optimization techniques, NEAT automatically learns an appropriate representation for the solution.

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.

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

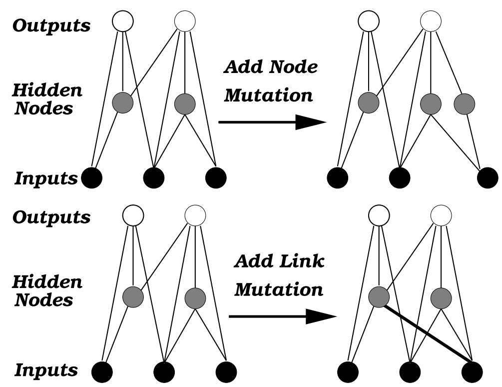


Figure 1: Examples of NEAT's structural mutation operators. At top, a new hidden node, shown on the right, is added to the network by splitting an existing link in two. At bottom, a new link, shown with a thicker black line, is added to connect two existing nodes.

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

Since NEAT is a general purpose optimization technique, it can be applied to a wide variety of problems. When applied to reinforcement learning problems, NEAT typically evolves *action selectors*, which directly map states to the action the agent should take in that state. Since it does not estimate value functions, it is an example of *policy search* reinforcement learning. Like other policy search methods, e.g. (Sutton *et al.* 2000; Kohl & Stone 2004), it uses global optimization techniques to directly search the space of potential policies. In the following section we describe how NEAT can be used to evolve Q-learning function approximators instead of action selectors.

Method

In this section, we describe NEAT+Q, the implementation of evolutionary function approximation used in our experi-

[1] This section is adapted from (Stanley & Miikkulainen 2002).

ments. We also describe sample-efficient NEAT+Q, an enhancement designed to reduce sample complexity.

NEAT+Q

As described above, when evolutionary methods are applied to reinforcement learning problems, they typically evolve a population of action selectors, each of which remains fixed during its fitness evaluation. The central insight behind evolutionary function approximation is that, if evolution is directed to evolve value functions instead, then those value functions can be updated, using TD methods, during each fitness evaluation. In this way, the system can *evolve* function approximators that are better able to *learn* via TD. In addition to automating the search for effective representations, evolutionary function approximation can enable synergistic effects between evolution and learning via a biological phenomenon called the *Baldwin Effect* (Baldwin 1896), which can speed up evolutionary computation (Hinton & Nowlan 1987; Ackley & Littman 1991). When each individual can learn during its lifetime, it need not be perfect at birth. Hence, the Baldwin Effect predicts that evolution will find good solutions more easily. In the remainder of this section, we describe NEAT+Q, a particular implementation of evolutionary function approximation.

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, they will effectively evolve value functions instead of action selectors. Hence, the outputs are no longer arbitrary values; they represent the long-term discounted values of the associated state-action pairs and are used, not just to select the most desirable action, but to update the estimates of other state-action pairs.

Algorithm 1 summarizes the resulting NEAT+Q method. Each time the agent takes an action, the network being evaluated is backpropagated once towards Q-learning targets (line 15) and the agent uses ϵ-greedy selection (Sutton & Barto 1998) to ensure it occasionally tests alternatives to its current policy (lines 12–13). If α and ϵ are set to zero, this method degenerates to regular NEAT. NEAT+Q maintains a running total of the reward accrued by the network during its evaluation (line 18). Each generation ends after e episodes, at which point each network's average fitness is $N.fitness/N.episodes$. NEAT creates a new population by repeatedly calling the BREED-NET function (line 23), which performs crossover on two highly fit parents. The new resulting network can then undergo mutations that add nodes or links to its structure (lines 24–25).

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

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

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: exploration rate*
5:
6: $P[] \leftarrow$ INIT-POPULATION(S, A, p)
7: **for** $i \leftarrow 1$ to g **do**
8:   **for** $j \leftarrow 1$ to e **do**
9:     $N, s, s' \leftarrow P[j \% p]$, null, INIT-STATE$(S)$
10:     **repeat**
11:       $Q[] \leftarrow c\times$ EVAL-NET(N, s')
12:       **with-prob**(ϵ) $a' \leftarrow$ RANDOM(A)
13:       **else** $a' \leftarrow \mathrm{argmax}_k Q[k]$
14:       **if** $s \neq$ null **then**
15:         BACKPROP$(N, s, a, r + \gamma \mathrm{max}_k Q[k]/c, \alpha, \gamma, \lambda)$
16:       $s, a \leftarrow s', a'$
17:       $r, s' \leftarrow$ TAKE-ACTION(a')
18:       $N.fitness \leftarrow N.fitness + r$
19:     **until** TERMINAL-STATE?(s)
20:     $N.episodes \leftarrow N.episodes + 1$
21:   $P'[] \leftarrow$ new array of size p
22:   **for** $j \leftarrow 1$ to p **do**
23:     $P'[j] \leftarrow$ BREED-NET$(P[])$
24:     **with-prob** m_n: ADD-NODE-MUTATION$(P'[j])$
25:     **with-prob** m_l: ADD-LINK-MUTATION$(P'[j])$
26:   $P[] \leftarrow P'[]$

Sample-Efficient NEAT+Q

For both NEAT and NEAT+Q, the number of episodes per generation e must be much greater than the population size $|P|$ in domains that are highly stochastic. Such domains have noisy fitness functions and hence each network's performance must be averaged over many episodes. For NEAT+Q, however, there is a second reason to set e high, which applies even if the domain is deterministic: Q-learning needs time to learn. In most domains, TD updates will not have substantial impact in a single episode.

Consequently, the original NEAT+Q method is likely to offer a practical advantage over regular NEAT only in highly stochastic domains, where e must be set high anyway. Otherwise, even if NEAT+Q ultimately discovers better policies, it will take many more episodes to do so. In this section, we present sample-efficient NEAT+Q, a variation designed to remedy this shortcoming. By training networks on saved experience, Q-learning can have a substantial impact even when $e = |P|$. As a result, NEAT+Q can improve performance even in completely deterministic domains.

Sample-efficient NEAT+Q works by exploiting the off-policy nature of Q-learning. Because Q-learning's update rule is independent of the policy the agent is following, one network can be updated while another is controlling the agent. Furthermore, a network can be updated based on data saved from previous sample episodes, regardless of what policy was used during those episodes. Consequently, it is not necessary to use different episodes to train each network. On the contrary, by saving data from the episodes used by the previous generation, each network in the population can be pre-trained, using computation time but no additional sample episodes. If the fitness function is not too noisy then, once trained, the resulting function approximators can be evaluated by NEAT+Q using only $|P|$ episodes.

To achieve this sample-efficiency, NEAT+Q records all the transition samples, of the form (s, a, r, s'), from the episodes used to evaluate the previous generation. Then, at the beginning of each generation (i.e. after line 7 in Algorithm 1), it calls the PRE-TRAIN function described in Algorithm 2. In the first generation no samples have been collected ($|T| = 0$) and no pre-training occurs.

Algorithm 2 PRE-TRAIN$(P, T, c, \alpha, \gamma, \lambda)$

// P: population, T: sample transitions, c: output scale, α: learning rate
// γ: discount factor, λ: eligibility decay rate

for $i \leftarrow 1$ to $|P|$ **do**
  for $j \leftarrow 1$ to $|T|$ **do**
    $Q[] \leftarrow$ EVAL-NET$(P[i], T[j].s')$
    BACKPROP$(P[i], T[j].s, T[j].a, T[j].r + \gamma \max_k Q[k]/c, \alpha, \gamma, \lambda)$

If computational resources are plentiful, there are many ways to extend the pre-training phase. For example, episodes could be saved from all previous generations instead of just the last one and/or each network could be trained repeatedly on each sample instead of just once. To make our experiments more feasible, we do not evaluate these alternatives in this paper. However, the experiments presented below suggest that additional pre-training does not improve performance.

Assuming e is reduced to $|P|$, this algorithm will have much higher amortized computational complexity per episode than the original NEAT+Q method, since each network must be trained before evaluations can begin. However, it will have much lower sample complexity since each generation requires many fewer episodes. This trade-off is likely to be advantageous in practice, since sample experience is typically a much scarcer resource than computation time.

Server Job Scheduling

To evaluate sample-efficient NEAT+Q, we use a challenging reinforcement learning task called server job scheduling (Whiteson & Stone 2006). This domain is drawn from the burgeoning field of autonomic computing (Kephart & 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.

In server job scheduling, a server such as a website's application server or database must determine in what order to process the jobs currently waiting in its queue. Its goal is to maximize the aggregate utility of all the jobs it processes. A *utility function* for each job type maps the job's completion time to the utility derived by the user (Walsh *et al.* 2004). The problem of server job scheduling becomes challenging when these utility functions are non-linear and/or the server must process multiple types of jobs. Since selecting a particular job for processing necessarily delays the completion of all other jobs in the queue, the scheduler must weigh difficult trade-offs to maximize aggregate utility.

Our experiments were conducted in a Java-based simulator. 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.

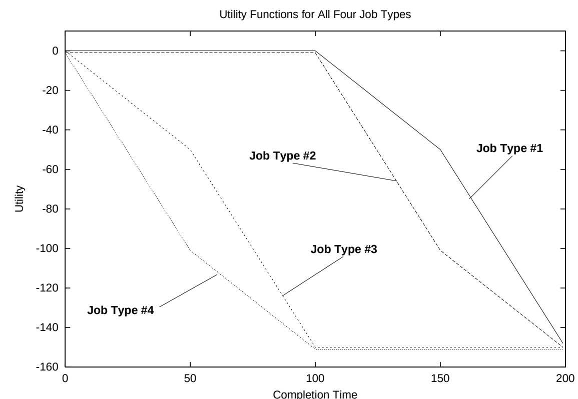


Figure 2: The four utility functions used in our experiments.

Utility functions for the four job types used in our experiments 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 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.

To render the state and action spaces more manageable, we discretize them. The range of job ages from 0 to 200 is divided into four equal 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.

In this paper, we consider a deterministic variation of the server job scheduling task. At the beginning of each learning run, we randomly select the sequence of 200 jobs that the agent will process in each episode. Hence, within each run, every episode uses the same sequence of jobs, though that sequence differs for each run. Making the task deterministic allows us to more effectively evaluate sample-efficient NEAT+Q. In the stochastic version of the task, the fitness function is very noisy and each network must be evaluated for approximately 100 episodes to get an accurate fitness estimate, giving the original NEAT+Q method enough time to significantly improve performance. In the deterministic version, each network can be accurately evaluated in a single episode and hence NEAT+Q will significantly improve performance only if it is made sample-efficient.

Results and Discussion

We begin by evaluating the behavior of the original NEAT+Q method in the server job scheduling task and comparing its performance to that of NEAT and Q-learning with manually designed neural networks. In our experiments, NEAT+Q uses the following parameter settings: c=-1000, p=50, m_n=0.01, m_l=0.05, α=0.1, γ=0.95, λ=0.0, ϵ=0.05. For each network in the population, α was linearly annealed at a rate of 1×10^{-3} per episode. NEAT used the same parameter settings but with α=0.0 and ϵ=0.0. For Q-learning with manually designed networks, we used the same setup that worked best in a previous study in this domain (Whiteson & Stone 2006). This setup, the best of 24 configurations tested, uses feed-forward networks with 4 hidden nodes, random weights drawn randomly from a Gaussian distribution with σ=0.1, α=0.01, γ=0.95, λ=0.6, ϵ=0.05. α was linearly annealed at a rate of 1×10^{-7} per episode. In all the results presented below, performance is averaged over 10 independent trials for each method.

Figure 3 highlights the advantages and disadvantages of NEAT+Q. In Figure 3a, both NEAT and NEAT+Q used 5,000 episodes per generation. Hence, each network was evaluated for 100 episodes. The fluctuations in the NEAT+Q graph consist of a series of intra-generational learning curves: the population's average performance improves gradually over the course of each generation due to Q-learning. The graph, which shows the uniform moving average score per episode over the past 1,000 episodes, clearly demonstrates that NEAT+Q can outperform both regular NEAT and Q-learning with manually designed networks. By automatically discovering effective representations for the function approximator, NEAT+Q outperforms Q-learning. By learning function approximators instead of fixed actions selectors and exploiting the Baldwin Effect, NEAT+Q also outperforms regular NEAT. A Student's t-test confirmed that the performance differences between these methods is statistically significant with 95% confidence.

However, since we are using the deterministic version of server job scheduling, it is not necessary to use so many episodes per generation. Though there is still slight noise in the fitness function due to ϵ-greedy action selection, each network can be accurately evaluated in a single episode. Figure 3b shows how the performance of NEAT and NEAT+Q change when they use only 50 episodes per generation. The graph shows uniform moving average score per episode averaged over the past 100 episodes. NEAT+Q's performance advantage disappears because Q-learning does not have a significant effect in one episode.

Nonetheless, NEAT+Q can substantially improve performance even in the deterministic version of the task if it is made sample-efficient. Figure 3b also shows the performance of sample-efficient NEAT+Q. By pre-training with saved episodes, this method substantially outperforms regular NEAT and the original NEAT+Q method. Obtaining this performance improvement requires additional computation time, but the result is a dramatic drop in sample complexity. By saving episodes, NEAT+Q can outperform NEAT even when the number of episodes per generation is reduced by two orders of magnitude. A Student's t-test confirmed that the performance difference between sample-efficient NEAT+Q and both regular NEAT+Q and NEAT is statistically significant with 95% confidence.

In these experiments, sample-efficient NEAT+Q saves transitions from all the episodes used to evaluate the previous generation. Hence, each network is pre-trained with 50 sample episodes. Would performance improve more if additional episodes were saved? Could the same performance be achieved with less computation time if fewer episodes were saved? To address these questions, we ran additional trials of sample-efficient NEAT+Q, pre-training on 5, 10, 25, or 100 episodes instead of 50. Figure 4 summarizes the results of these experiments by comparing the average performance of each method after 30,000 episodes. Surprisingly, pre-training with as few as 5 saved episodes still yields a substantial performance advantage. Furthermore, pre-training with 100 episodes does not improve performance. A Student's t-test demonstrated that, while the differences between each sample-efficient version of NEAT+Q and both regular NEAT+Q and NEAT are significant, the differences among them are not significant.

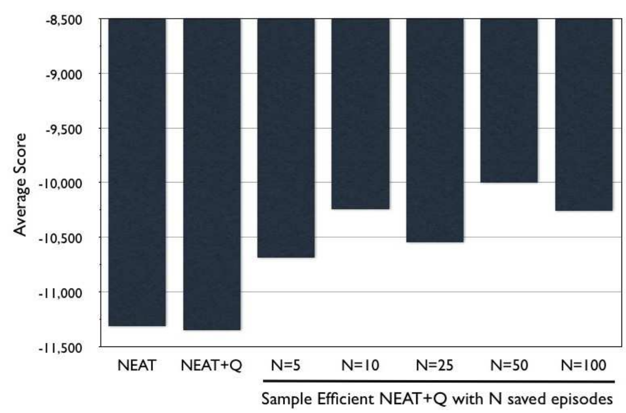
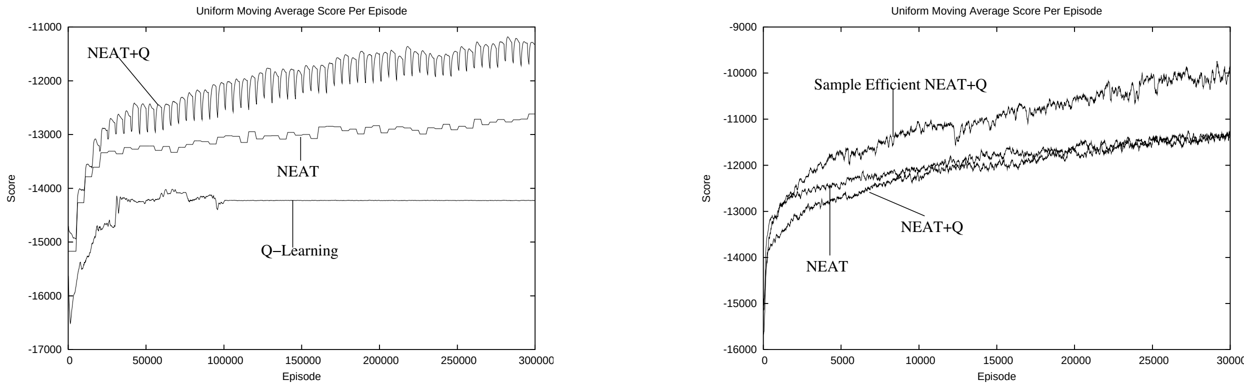


Figure 4: A comparison of the performance of NEAT, NEAT+Q and sample-efficient NEAT+Q with different numbers of saved episodes. Each bar represents the average score after 30,000 episodes and hence is comparable to the right edge of Figure 3b.

Together, these results clearly demonstrate that sample-efficient evolutionary function approximation can substantially improve performance. Furthermore, they can do so without using more sample episodes than a traditional evolutionary approach. The price for this sample efficiency is increased computational complexity. In practice, this trade-off is likely to be beneficial, since sample experience is typically a much scarcer resource. Even when computational resources are limited, this approach can be useful, as our results demonstrate that even a modest amount of pre-training can significantly improve performance.



(a) 5,000 episodes per generation

(b) 50 episodes per generation

Figure 3: A comparison of the original NEAT+Q method with regular NEAT and Q-learning with manually designed networks. In (a) both NEAT and NEAT+Q use 5,000 episodes per generation. In (b) they both use 50 episodes per generation.

Related Work

Because it saves sample episodes for reuse, sample-efficient NEAT+Q bears a close resemblance to experience replay methods for reinforcement learning (Lin 1992). In particular, it is similar to Neural Fitted Q Iteration (Reidmiller 2005), which uses data from saved episodes to train neural network TD function approximators. The primary difference is that these methods do not learn representations because they use saved experience to train only one function approximator. By contrast, sample-efficient NEAT+Q uses saved experience to train an entire population of function approximators with heterogeneous representations and then subjects them to evolutionary selection.

By offering a trade-off between computational and sample complexity, this method is similar in spirit to model-based reinforcement learning methods, e.g. (Moore & Atkeson 1993). Model-based methods strive to learn a model of the agent's environment, which makes it possible to update the value function off-line using dynamic programming. How much updating is performed typically depends on the amount of computation time available between each action. Implicit in the model-based approach is the assumption that all available computation time should be used to update a single value function. The success of sample-efficient evolutionary function approximation suggests that, when computational resources are plentiful, they can be used to optimize the agent's function approximator representation as well.

Conclusion

This paper presents an enhancement to evolutionary function approximation that, by exploiting the off-policy nature of certain TD methods, makes it much more sample-efficient and hence of practical value even in domains that are not highly stochastic. A particular implementation called sample-efficient NEAT+Q improves the performance of Q-learning, without using additional sample episodes, by automatically discovering appropriate topologies for pre-trained neural network function approximators. Empirical results in the server job scheduling domain demonstrate that the enhanced method can learn better policies than evolution or TD methods alone and can do so in many fewer episodes than standard evolutionary function approximation.

References

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

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

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

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

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

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

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

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

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

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

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

Moore, A. W., and Atkeson, C. G. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning* 13(1):103–130.

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

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

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

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

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

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

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

Watkins, C. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge.

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

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