Auton Agent Multi-Agent Syst
DOI 10.1007/s10458-009-9100-2

Critical factors in the empirical performance of temporal difference and evolutionary methods for reinforcement learning

Shimon Whiteson · Matthew E. Taylor · Peter Stone

Abstract Temporal difference and evolutionary methods are two of the most common approaches to solving reinforcement learning problems. However, there is little consensus on their relative merits and there have been few empirical studies that directly compare their performance. This article aims to address this shortcoming by presenting results of empirical comparisons between Sarsa and NEAT, two representative methods, in mountain car and keepaway, two benchmark reinforcement learning tasks. In each task, the methods are evaluated in combination with both linear and nonlinear representations to determine their best configurations. In addition, this article tests two specific hypotheses about the critical factors contributing to these methods' relative performance: (1) that sensor noise reduces the final performance of Sarsa more than that of NEAT, because Sarsa's learning updates are not reliable in the absence of the Markov property and (2) that stochasticity, by introducing noise in fitness estimates, reduces the learning speed of NEAT more than that of Sarsa. Experiments in variations of mountain car and keepaway designed to isolate these factors confirm both these hypotheses.

Keywords Autonomous agents · Reinforcement learning · Temporal difference learning · Evolutionary computation

This paper significantly extends an earlier conference paper, presented at the 2006 GECCO conference [72].

S. Whiteson (✉)
Informatics Institute, University of Amsterdam, Science Park 107, 1098 XG Amsterdam, The Netherlands
e-mail: s.a.whiteson@uva.nl

M. E. Taylor
Computer Sciences Department, The University of Southern California, 941 W. 37th Place, Los Angeles, CA 90089-0781, USA
e-mail: taylorm@usc.edu

P. Stone
Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX 78712-0233, USA
e-mail: pstone@cs.utexas.edu

Published online: 17 July 2009

1 Introduction

In the development of autonomous agents, *reinforcement learning* [69] has emerged as an important tool for discovering *policies* for sequential decision tasks. Unlike supervised learning, reinforcement learning assumes that examples of correct and incorrect behavior are not available. However, unlike unsupervised learning, it assumes that a reward signal can be perceived. Since many challenging and realistic tasks fall in this category, e.g., elevator control [15], helicopter control [47], and autonomic computing [75,79], developing effective reinforcement learning algorithms is crucial to the progress of autonomous agents.

The most well-known approach to solving reinforcement learning problems is based on *value functions* [9], which estimate the long-term expected reward of each state the agent may encounter, given a particular policy. If a complete model of the environment is available, dynamic programming [10] can be used to compute an optimal value function, from which an optimal policy can be derived. If a model is not available, one can be learned from experience [26,44,65,68]. Alternatively, an optimal value function can be discovered via model-free techniques such as temporal difference (TD) methods [67], which combine elements of dynamic programming with Monte Carlo estimation [5]. Currently, TD methods are among the most commonly used approaches for reinforcement learning problems.

However, reinforcement learning problems can also be tackled without learning value functions, by directly searching the space of potential policies. Evolutionary methods [46, 60,82], which simulate the process of Darwinian selection to discover highly fit policies, are one effective way of conducting such a search.

Unfortunately, there is little consensus on the relative merits of these two approaches to reinforcement learning. Evolutionary methods have fared better empirically on certain benchmark problems, especially those where the agent's state is only partially observable [20,21,46, 60]. However, value function methods typically have stronger theoretical guarantees [30,37]. Evolutionary methods have also been criticized because they do not exploit the specific structure of the reinforcement learning problem. As Sutton and Barto [69, Sect. 1.3] write, "It is our belief that methods able to take advantage of the details of individual behavioral interactions can be much more efficient than evolutionary methods in many cases".

Despite this debate, there have been surprisingly few studies that directly compare these methods. Those that do (e.g., [21,45,49,56,80]) rarely isolate the factors critical to the performance of each method. As a result, there are currently few general guidelines describing the methods' relative strengths and weaknesses. In addition, since the evolutionary and TD research communities are largely disjoint and often focus on different applications, there are no commonly accepted benchmark problems or evaluation metrics.

This article takes a step towards filling this void by presenting the results of an empirical study comparing Sarsa [55,66] and NEAT [60], two popular and empirically successful TD and evolutionary methods, respectively. No empirical study can ever be comprehensive in the methods it evaluates or the testbeds it employs. This study instead focuses on comparing these representative methods in two domains: mountain car [12], a well-known benchmark problem, and keepaway [63], a challenging robot soccer task with noisy sensors and complex, stochastic dynamics. In each task, the methods are evaluated in combination with both linear and nonlinear representations of their policies or value functions in order to determine their best configurations.

This article's experiments contribute to a body of empirical comparisons between TD and evolutionary methods that is much in need of expansion. These works help address questions about when each method is preferable. However, they do little to explain *why* these methods perform as they do. To address this shortcoming, we formulate specific hypotheses about the

factors critical to each method's performance and devise variations of the two domains that are designed to test them. In particular, we propose the following two hypotheses:

1. Sensor noise reduces the final performance of Sarsa more than that of NEAT since Sarsa, like other TD methods, relies on an update rule that assumes access to Markovian state information. By contrast, NEAT simply searches the space of policies, making no such assumption.
2. Stochasticity, by introducing noise in fitness estimates, reduces the learning speed of NEAT more than that of Sarsa. Compensating for this noise requires performing longer fitness evaluations, greatly slowing evolution's progress. By contrast, Sarsa requires at worst a lower learning rate and can even be aided by stochasticity, which provides a natural form of exploration.

We test these hypotheses by conducting empirical comparisons on variations of mountain car and keepaway where sensor noise and/or stochasticity have been added or removed. The results confirm that these factors are indeed critical to each method's performance, since varying the domains in these ways causes dramatic changes in the relative performance of the two methods.

The remainder of this paper is organized as follows. Section 2 overviews the NEAT and Sarsa methods and Sect. 3 describes the mountain car and keepaway tasks. Section 4 presents empirical results on the benchmark versions of these tasks. Sections 5 and 6 present the results of experiments that isolate the effects of sensor noise and stochasticity, respectively, in each domain. Section 7 reviews related work, Sect. 8 outlines ideas for future work, and Sect. 9 concludes.

2 Methods

The goal of this article to provide useful empirical comparisons between TD and evolutionary methods for RL. Therefore, to keep the scope of the article focused, we do not consider other policy search approaches, e.g., gradient methods [3,7,34,70] or other value function approaches, e.g., model-based methods [14,30,65]. (See Sect. 8 for a more complete discussion of additional comparisons that would be useful to conduct in the future.)

Even given a focus on TD and evolutionary methods, there are a wide variety of methods in use today from which we can choose. No single empirical study can hope to include them all. In this article, we focus on two well-known, representative methods: Sarsa and NEAT. We believe these methods are appropriate choices for two reasons. First, we have substantial experience using these methods. In addition to the obvious practical advantages, this familiarity enables us to set both algorithms' parameters with confidence. Second, these methods are often used in practice. This is important because our goal is to assess the strengths and weaknesses of methods that are currently in common usage. Hence, our choice of methods does not necessarily imply they are the best available, but merely that they are popular. Nonetheless, there is considerable evidence that both Sarsa and NEAT are well-suited to the tasks we consider [64,66,78,79]. Furthermore, we strive to configure these methods with the best input representation and approximation architecture for each task, either by reference to previous literature on their application to the given domain or by conducting our own comparisons of different configurations (see Sect. 4 for details). In the remainder of this section, we provide some background on the Sarsa and NEAT algorithms.

2.1 Sarsa

Many reinforcement learning methods rely on the notion of value functions, which estimate the long-term expected reward of each state the agent may encounter, given a particular policy. If the state space is finite and the agent has a complete model of its environment, then the optimal value function, and therefore an optimal policy, can be computed using dynamic programming [10]. Dynamic programming estimates the value of each state by exploiting its close relationship to the value of those states which might occur next. By repeatedly iterating over the state space and updating these estimates, dynamic programming can compute the optimal value function.

However, dynamic programming is not directly applicable when a complete model of the environment is not available. Fortunately, the optimal value function can be learned without a model using TD methods [67], which synthesize dynamic programming with Monte Carlo methods. TD methods use the agent's immediate reward and state information to update the value function.

One way of performing such updates is via the Sarsa method. Sarsa is an acronym for State Action Reward State Action, describing the 5-tuple needed to perform the update: (s, a, r, s', a'), where s and a are the agent's current state and action, r is the immediate reward the agent receives from the environment, and s' and a' are the agent's subsequent state and chosen action. In the simple case, the value function is represented in a table, with one entry for each state-action pair. After each action, the table is updated according to the following rule:

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

where α is the learning rate and γ is a discount factor weighting immediate rewards relative to future rewards.

Like dynamic programming, Sarsa estimates the value of a given state-action pair by bootstrapping off estimates of other such pairs. In particular, the value of a given state-action pair (s, a) can be estimated as $r + \gamma Q(s', a')$, which is the discounted value of the subsequent state-action pair (s', a') plus the immediate reward received during the transition. Sarsa's update rule takes the old value estimate $Q(s, a)$, and moves it incrementally closer to this new estimate. The learning rate α controls the size of these adjustments. As these value estimates become more accurate, the agent's policy will improve.

Since a model is not available, Sarsa cannot simply iterate over all state-action pairs to perform updates. Instead, the agent can only perform updates based on transitions and rewards it observes while interacting with its environment. Thus, it is critical that the agent visits a broad range of states and tries various actions if it is to discover a good policy. To achieve this, TD methods are typically coupled with exploration mechanisms which ensure that the agent, rather than always behaving greedily with respect to its current value function, sometimes tries alternative actions. One simple exploration mechanism is called ϵ-greedy exploration [76], whereby the agent takes a random action at each time step with probability ϵ, and takes the greedy action otherwise. Often, ϵ is annealed over time by multiplying it by a decay rate $d \in [0, 1]$ after each episode.

While the value function can be represented in a table in simple tasks, this approach is infeasible for most real-world problems because the state space grows exponentially with respect to the number of state features, a problem Bellman [10] dubbed the "curse of dimensionality". Hence, the agent may be unable even to store such a table, much less learn correct values for each entry in reasonable time. Moreover, many problems have continuous state

features, in which case the state space is infinite and a table-based approach is impossible even in principle.

In such cases, TD methods rely on function approximation. In this approach, the value function is not represented exactly but instead approximated via a parameterized function. Typically, those parameters are incrementally adjusted via supervised learning methods to make the function's output more closely match estimated targets generated from the agent's experience. Many different methods of function approximation have been used successfully. In this paper, we couple Sarsa with *tile coding* [1], *radial basis function approximators* (RBF) [51], and neural networks [2]. In the case of linear function approximation, the update rule specified in Eq. 1, is replaced by the following:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha[r + \gamma Q(s', a') - Q(s, a)]\Delta_{\boldsymbol{\theta}} Q(s, a)$$

where $\boldsymbol{\theta}$ is the vector of weight values being learned and $\Delta_{\boldsymbol{\theta}} Q(s, a)$ is the gradient of $Q(s, a)$ with respect to $\boldsymbol{\theta}$.

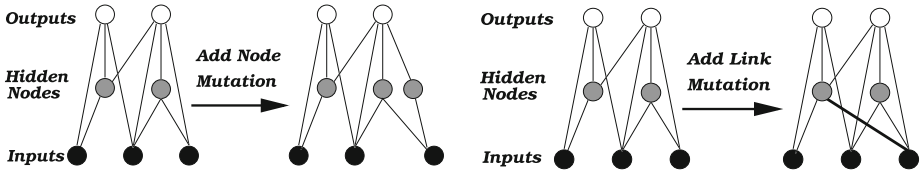
2.2 NeuroEvolution of augmenting topologies (NEAT)

Policy search methods do not explicitly reason about value functions but instead use optimization techniques to directly search the space of policies for one that accrues maximal reward. To assess the performance of each candidate policy, the agent typically employs the policy for one or more episodes and sums the total reward received.

Among the most successful approaches to policy search is neuroevolution [82], which uses evolutionary computation [18] to optimize a population of neural networks. In a typical neuroevolutionary system, the weights of a neural network are concatenated to form an individual *genome*. A *population* of such genomes is then evolved by repeatedly evaluating each genome's *fitness* and selectively reproducing the best ones. Fitness is measured with a domain-specific *fitness function*; in reinforcement learning tasks, the fitness function is typically the average reward received during some number of episodes in which the agent employs the policy specified by the given genome. The fittest individuals are used to breed a new population via *crossover* and *mutation*. Most neuroevolutionary systems require the designer to manually determine the network's representation (i.e., how many hidden nodes there are and how they are connected).

However, some neuroevolutionary methods can automatically evolve representations along with network weights. In particular, NeuroEvolution of augmenting topologies (NEAT) [60] combines the usual search for network weights with evolution of the network structure. Unlike other systems that evolve network topologies and weights [22,82], 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 are likely 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. The remainder of this section provides an overview of NEAT's reproductive process. Stanley and Miikkulainen [60] present a full description.

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,



(a) A mutation operator for adding new nodes **(b)** A mutation operator for adding new links

Fig. 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

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 two 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 cross over, 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 [53] is essentially avoided.

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 species 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. Genomes are tested one at a time and if its distance to a randomly chosen member of the species is less than 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* [18], where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

In reinforcement learning tasks, NEAT typically evolves *action selectors*, which have one or more inputs for each state feature and one output for each action; the agent takes the action whose corresponding output has the highest activation. However, since the network represents a policy, not a value function, the activations on the output nodes do not represent value estimates. In fact, the outputs can have arbitrary activations so long as the most desirable action has the largest activation. If the domain is noisy, the reward accrued in a single episode may be unreliable, in which case obtaining accurate fitness estimates requires *resampling*, i.e., averaging performance over several episodes. NEAT has proven particularly effective in reinforcement learning domains, amassing empirical successes on several difficult tasks like non-Markovian double pole balancing [60], robot control [61], and autonomic computing [79].

Note that while evolutionary methods like NEAT are sometimes parallelized to improve their computational efficiency, doing so is not feasible in reinforcement learning tasks. Unless the agent learns a model of the world, estimating a policy's fitness requires executing it in the environment, which can only be done serially. Thus evaluating a population of size 100 takes twice as many episodes as evaluating a population size of 50, and 100 times as long as updating a value function with Sarsa for one episode. Of course, for the domains considered in this article, the environment is itself a computer program so in principle evolutionary fitness evaluations could be parallelized when conducting experiments, so long as the method is still "charged" for each episode when reporting results. For reasons of simplicity, fitness evaluations are conducted serially in our experiments.

3 Domains

In this article we compare Sarsa and NEAT on two reinforcement learning problems, mountain car and keepaway, and variations thereof. There are several reasons for selecting these tasks.

Mountain car is a classic benchmark problem, perhaps the most well-known of all reinforcement learning problems. As a result, effective strategies for applying both TD and evolutionary methods are already known. Thus, we can conduct experiments with high confidence that the results reflect the full potential of each method. Furthermore, the simplicity of the task makes it feasible to conduct large numbers of experiments and obtain truly comprehensive results.

Due to the great interest in RoboCup soccer (e.g., the 2005 World Championships in Osaka, Japan attracted 180,000 spectators), keepaway has also become an important benchmark task. Since the task involves multiple agents, a large state space, and noisy sensors and effectors, it is more complex and realistic than most reinforcement learning benchmark problems. Hence, it allows us to evaluate the ability of NEAT and Sarsa to scale up to more challenging tasks.

The remainder of this section introduces the mountain car and keepaway tasks and describes how Sarsa and NEAT are applied to them in our experiments.

3.1 Mountain car

In the mountain car task [12], depicted in Fig. 2, the agent's goal is 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 momentum to ascend to the goal before running out of speed.

The agent's state at time step t consists of its current position x_t and velocity $\dot{x}_t$. It receives a reward of -1 at each time step until reaching the goal ($x_t \geq 0.5$), at which point the episode terminates. The agent's action $a_t \in \{1, 0, -1\}$ corresponds to one of three available throttle settings: forwards, neutral, and backwards. The following equations control the car's movement:

$$x_{t+1} = x_t + \dot{x}_{t+1}$$
$$\dot{x}_{t+1} = \dot{x}_t + 0.001a_t - 0.0025\cos(3x_t)$$

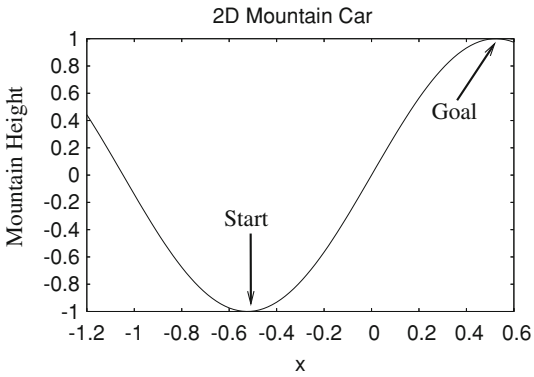


Fig. 2 The mountain car task, in which an underpowered car strives to reach the top of a mountain

Position and velocity are constrained such that $-1.2 \leq x_t \leq 0.6$ and $-0.07 \leq \dot{x}_t \leq 0.07$. In each episode, the agent begins in a state chosen randomly from these ranges. If the agent's position ever becomes -1.2, its velocity is reset to zero. To prevent episodes from running indefinitely, each episode is terminated after 5,000 steps if the agent still has not reached the goal.

3.1.1 Applying Sarsa to mountain car

Despite the apparent simplicity of mountain car, solving it with TD methods requires function approximation, since its state features are continuous. Previous research has demonstrated that TD methods can solve mountain car using several different function approximators, including tile coding [35,66], locally weighted regression [12], decision trees [52], radial basis functions [35], and instance-based methods [12]. In this work, we evaluate three ways of approximating the agent's value function: tile coding, single-layer perceptrons and multi-layer perceptrons.

In the first approach, *tile coding* [1], a piecewise-constant approximation of the value function is represented by a set of exhaustive partitions of the state space called *tilings*. Typically, the tilings are all partitioned in the same way but are slightly offset from each other. Each element of a tiling, called a *tile*, is a binary feature activated if and only if the given state falls in the region delineated by that tile. Figure 3 illustrates a tile-coding scheme with two tilings.

Each tile has a weight associated with it and the value function for a given state is simply the sum of the weights of all activated tiles. The weights of the tile coding are learned via TD updates.

Consistent with previous research in this domain [66], we employ separate tile codings for each of the three actions: each tile coding independently learns to predict the action-value function for its corresponding action. Each tile coding uses 14 tilings, evenly spaced, and a tiling consists of a 9×9 grid of equally sized tiles.[1] Tile weights are learned using Sarsa with ϵ-greedy exploration.

In the second approach, *single-layer perceptrons* (SLPs), feed-forward neural networks without any hidden nodes, are used to represent a linear approximation of the agent's value function. We employ a typical formulation, where the input nodes describe the agent's current

[1] Our implementation uses Richard Sutton's Tile Coding Software version 2.0, available at http://www.cs.ualberta.ca/~sutton/tiles2.html.

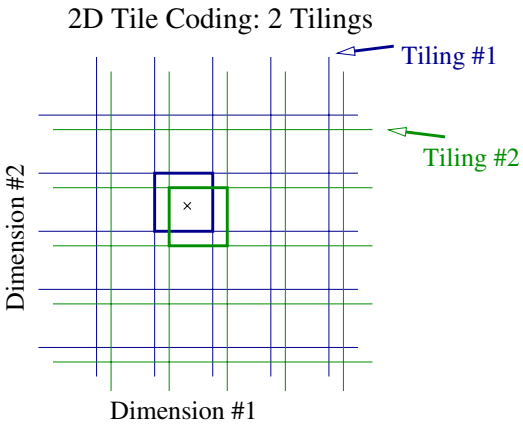


Fig. 3 An example of tile coding with two tilings. *Thicker lines* indicate which tiles are activated for the given state, marked with an 'x'

state and the outputs, one for each action, represent estimates of the value of the corresponding state-action pair. Since there are no hidden nodes, one completely connected layer of weights lies between the input and output nodes. In mountain car, an obvious choice of input representation is to use two real-valued inputs, one for the agent's position and one for its velocity. In this article, we also consider an expanded representation that uses 20 binary inputs. Each state feature is divided into ten equally-sized regions and one input is associated with each region.[2] That input is set to 1.0 if the agent's current state falls in that region and to zero otherwise. Hence, only two inputs are activated for any given state. Previous research [79] has shown that this expanded representation improves the performance of NEAT in mountain car. We consider it also for Sarsa to ensure that state representation is not a confounding factor in our results.

In the third approach, *multi-layer perceptrons* (MLPs), which are feed-forward neural networks containing hidden nodes, are used to represent a nonlinear approximation of the agent's value function. Such networks have greater representational power than SLPs, though learning the correct weights can be more difficult. We consider only networks with a single layer of hidden nodes, such that the inputs are completely connected to the hidden nodes and the hidden nodes are completely connected to the outputs. As with SLPs, we consider two input representations for mountain car, one with two real-valued inputs and one with 20 binary inputs.

3.1.2 Applying NEAT to mountain car

For the mountain car task, NEAT is used to evolve a population of neural networks, each of which represents a policy (i.e., it maps states to actions). As with Sarsa, we consider both the 2-input representation and the expanded 20-input representation. In both cases, the neural networks have three output nodes, one per action, and the output node with the highest activation dictations the action chosen for the current input state. We also evaluate the performance of NEAT when structural mutations are completely disabled and when they are allowed. In the former case, NEAT evolves only the weights of a population of SLPs. Hence, the space of policies it searches is restricted to linear functions. In the latter case, structural mutations can result in the addition of hidden nodes, allowing the representation of nonlinear policies.

[2] For example, the velocity state variable ranges from -0.07 to 0.07, and thus the ten regions are $[-0.07, -0.056), [-0.056, -0.042), \ldots, [0.056, 0.07]$.

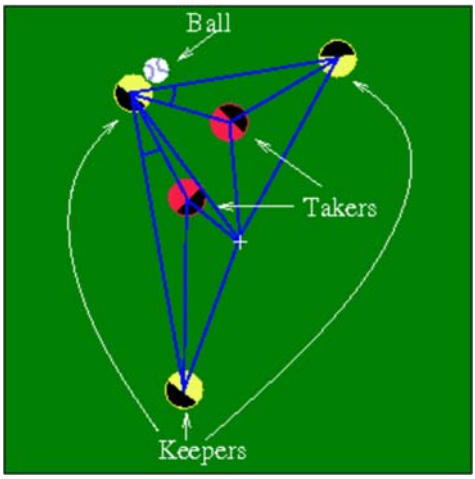


Fig. 4 13 State variables are used for learning with three keepers and two takers. The state is egocentric and rotationally invariant for the keeper with the ball; there are 11 distances, indicated with *straight lines*, between players and the center of the field as well as two angles along passing lanes

3.2 Keepaway

Keepaway is a simulated robot soccer task built on the RoboCup Soccer Server [48], an open source software platform that has served as the basis of multiple international competitions and research challenges. The server simulates a complete 11 versus 11 soccer game in which each player employs unreliable sensors and actuators. In particular, the perceived distance to objects is quantized and uniformly distributed noise is added to all objects' movements. Stone [62, Chap. 2] provides a complete description of the simulator's dynamics, including sensor and actuator noise.

Keepaway is a subproblem of the full simulated soccer game in which a team of three *keepers* attempts to maintain possession of the ball on a 20 m $\times$ 20 m field while two *takers* attempt to gain possession of the ball or force it out of bounds, ending the episode.[3]

Three keepers are initially placed in three corners of the field and a ball is placed near one of them. Two takers are placed in the fourth corner. When an episode starts, the keepers attempt to maintain control of the ball by passing among themselves and moving to open positions. The agent's state is defined by 13 variables, as shown in Fig. 4. The episode finishes when a taker gains control of the ball or the ball is kicked out of bounds. The episode is then reset with a random keeper placed near the ball. The initial state is different in each episode because the same keeper does not always start in the same corner and because the keepers are only placed near the corners rather than in exact locations.

The agents choose not from the simulator's primitive actions but from a set of higher-level macro-actions implemented as part of the player. These macro-actions can last more than one time step and the keepers make decisions only when a macro-action terminates. The macro-actions are *holdBall*, *pass*, *getOpen*, and *receive* [64]. The first two action are available only when the keeper is in possession of the ball; the latter two are available only when it is not. The *pass* action can be directed towards either of the keeper's teammates.

The agents make decisions at discrete time steps, at which point macro-actions are initiated and terminated. The reward for a macro-action is the number of time steps until the

[3] Experiments in this article use soccer server version 9.4.5 and version 0.5 of the benchmark keepaway implementation [63], available at http://www.cs.utexas.edu/~AustinVilla/sim/Keepaway/.

agent can select a new macro-action, or until the episode terminates.[4] Takers do not learn and always follow a static hand-coded strategy; both takers directly charge the ball as two takers are needed to capture the ball from a single keeper.

The keepers learn in a constrained policy space: they have the freedom to decide which action to take only when in possession of the ball. A keeper in possession of the ball may either hold it or pass it to one of its teammates, i.e., its action space is {*hold*, *passToTeammate1*, *passToTeammate2*}. Keepers not in possession of the ball execute a fixed strategy in which the keeper that can reach the ball fastest executes the *receive* macro-action and the remaining players execute the *getOpen* macro-action.

3.2.1 Applying Sarsa to keepaway

We use Sarsa to train teams of heterogeneous agents, with each keeper independently updating its own value function. Since Sarsa's learning rule is applied after each action, this approach is simpler than learning teams of homogeneous agents, which would require each agent to update the same value function. Doing so would be infeasible because communication bandwidth between the agents is limited and degrades with their relative distance. Since learners must select from macro-level actions that may take multiple time-steps, we use a SMDP [13] version of Sarsa, as in previous keepaway research [64], combined with ϵ-greedy exploration.

Due to the computational expense of conducting experiments in the keepaway domain (see details of training times in Sect. 4.2), we do not compare Sarsa using multiple input representations and function approximators as we do in mountain car. Instead, we employ only the best performing configuration previously reported in the literature. Specifically, to approximate the value function, we use a radial basis function approximator (RBF) [51], as a previous study showed that it was superior to tile coding in keepaway [63]. The same study also showed that RBFs perform better than neural network approximators even though the latter are capable of representing more complex, nonlinear functions.

Like tile coding, RBFs estimate the value function as the weighted sum of a set of features. Unlike tile coding, those features are not binary but lie in the interval [0, 1]. The ith feature f_i has a center c_i corresponding to a point in the state space. The value of the feature for a given state is some function, typically Gaussian, of the distance between the center and that state. As with tile coding in mountain car, the agent learns separate value functions for each action in keepaway. Following the model of previous research [63,64], we also treat each state feature separately, summing values for 13 independent RBFs. As shown in Fig. 5, we set the features to be evenly spaced Gaussian functions, where

$$f(x) = \exp\left(-\frac{|x - c_i|^2}{2\sigma^2}\right) \tag{2}$$

The σ parameter controls the width of the Gaussian function and therefore the amount of generalization over the state space. In keepaway, we use the previously established value of $\sigma = 0.25$. For each feature, there are 32 tilings of two tiles each, and the c_is are evenly spaced across each state variable range.

3.2.2 Applying NEAT to keepaway

As in mountain car, we use NEAT to evolve a population of networks that represent policies, using a setup previously reported to perform well in this domain [72]. NEAT uses the default

[4] This is equivalent to providing the keepers with a reward of +1 for every time step that the ball remains in play.

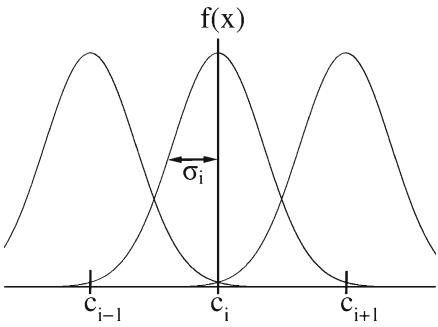


Fig. 5 An RBF approximator computes $Q(s, a)$ via a weighted sum of Gaussian functions. The contribution from the ith Gaussian is weighted by the distance from its center, c_i, to the relevant state variable. σ can be tuned to control the width of Gaussians and thus how much the function approximator generalizes

parameter settings with structural mutations turned on (see the "Appendix" for details) and each network has 13 inputs, corresponding to the 13 keepaway state variables, and 3 outputs, corresponding to every available macro-action. We use NEAT to evolve teams of homogeneous agents: in any given episode, the same neural network controls all three keepers on the field. The reward accrued during that episode then contributes to NEAT's estimate of that network's fitness. While heterogeneous agents could be evolved using cooperative coevolution [50], doing so is beyond the scope of this article.[5]

Since the keepaway task is highly stochastic, resampling is essential. One difficult question is how to distribute evaluation episodes among the organisms in a particular generation, given a noisy fitness function. While previous researchers have developed statistical schemes for performing such allocations [8,59], in this paper we adopt a simple heuristic strategy to increase the performance of NEAT: we concentrate evaluations on the more promising organisms in the population because their offspring will populate the majority of the next generation. In each generation, we conduct 6,000 evaluations.[6] Every organism is initially evaluated for ten episodes. After that, the highest ranked organism that has not already received 100 episodes is always chosen for evaluation. This process repeats until all 6,000 evaluations have been completed. Hence, every organism receives at least 10 evaluations and no more than 100, with the more promising organisms receiving the most.

4 Benchmark results

We begin our empirical analysis by comparing Sarsa and NEAT in the benchmark versions of both the mountain car and keepaway tasks. The differences observed in these experiments are used to formulate specific hypotheses about the critical factors of each method's performance. Those hypotheses are presented and tested in Sects. 5 and 6.

We evaluate the algorithms in an on-line setting, i.e., assuming each learning agent is situated in the environment and receives state and reward feedback after each action it takes. Thus, the agent cannot request samples from arbitrary states, but can learn only from

[5] The fact that Sarsa trains heterogeneous agents while NEAT trains homogeneous ones might appear to give NEAT an unfair advantage, since learning three policies is presumably harder than learning one. However, in informal experiments we found that Sarsa's performance does not improve when inter-agent communication is artificially allowed and Sarsa is used to train homogeneous teams. To be consistent with previous literature [63,64], we present results only on the communication-free version of the task.

[6] Preliminary tests found that 6,000 evaluations per generation results in superior performance than either 1,000 or 10,000 evaluations per generation.

samples gathered during its on-line experience, a scenario sometimes called an *on-line simulation model* [27].

In order to compare Sarsa and NEAT, we need a way to measure the quality and speed of learning for each method. In other words, we need to measure the quality of the best policy each method has discovered so far at various points in the learning process. For Sarsa, this is just the greedy policy ($\epsilon = 0.0$) that corresponds to the agent's current estimate of the value function. For NEAT, it is the champion of the most recently completed generation.[7]

Since fitness evaluations can be noisy and Sarsa uses exploration ($\epsilon \neq 0.0$) while learning, the quality of the best policy at a given point cannot be definitely established from each method's performance during learning. Instead, we assess the policies in retrospect by conducting additional evaluations after the learning runs have completed. After NEAT agents finish learning, we select the champion from each generation and evaluate it for 1,000 episodes. For Sarsa, we utilize the estimated value function at 1,000 episode intervals and evaluate the corresponding greedy policy, without learning, for 1,000 episodes.

Note that these measurements consider only the performance of the best policies discovered by each method at various points in the learning process; we do not measure other factors such as the computational or space requirements of each method. We focus on this performance metric for two reasons. First, the other factors are less critical in many real-world problems, wherein computational resources are often plentiful but interacting with the environment to gain experience for learning is expensive and dangerous. Second, the computational and space requirements of the algorithms we consider are relatively modest. For example, the computational requirements of Sarsa and NEAT are much lower than in many model-based approaches to RL [14,30,65].

4.1 Mountain car

Before comparing Sarsa and NEAT in mountain car, we first determine the best configuration for each method. For Sarsa, we compare the different function approximators described in Sect. 3.1.1. For the neural network function approximators, we consider input representations using either two or 20 inputs. For NEAT, we compare performance with or without structural mutations and using either the 2-input or 20-input representations.

The results of the Sarsa comparisons are shown in Fig. 6 (see the "Appendix" for details regarding learning parameters used in this comparison). In this and subsequent graphs, error bars represent the standard deviation over all evaluations of learning trials: each of the 50 learning trials is evaluated off-line for 1,000 episodes (after various amounts of learning), and we then graph the average and standard deviation of these 50 data. These results clearly demonstrate that tile coding is a better choice of function approximator for this task than neural networks, as it greatly outperforms all of the neural network alternatives. While tile coding quickly discovers excellent policies, none of the neural network configurations are able to achieve good performance. This result may seem surprising, but it is consistent with previous literature on the mountain car problem, as several researchers have noted that value estimates generated with neural networks using the 2-input representation can easily diverge [12,52]. To our knowledge, Sarsa has never been previously tested with neural networks using the 20-input representation. However, Q-learning [76], a TD method similar to Sarsa, has been

[7] In theory, it is possible that these are not the best policies discovered so far. Since Sarsa is an *on-policy* TD method, the greedy policy could perform worse than the exploratory one. It is also possible that the current generation champion in NEAT is inferior to a previous generation champion. However, we find that such differences are negligible in practice.

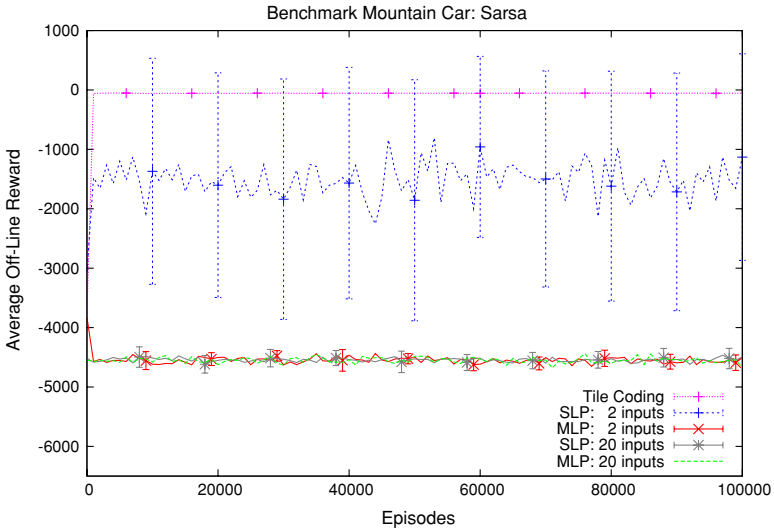


Fig. 6 A comparison of the average reward of the policies discovered by Sarsa using different function approximators and input representations in the benchmark mountain car task

tested with such networks and achieved similarly poor performance, except when combined with an evolutionary method that discovers a suitable network topology and initial weights [79]. Since we test only two network topologies, we cannot rule out the possibility that there exists a topology which performs better than tile coding. However, identifying such a scenario would require substantial engineering of the network structure. Previous research has shown that, in the case of Q-learning, even an extensive search for the right topology does not yield high-performing neural network function approximators for this task [79].

The results of the NEAT comparisons are shown in Fig. 7 (see the "Appendix" for details about all learning parameters used in the comparison). In this and subsequent graphs, error bars represent the standard deviation over all evaluations of learning trials: the champion of each of the 50 learning trials (after various amounts of training) is evaluated off-line for 1,000 episodes, and we then graph the average and standard deviation of these 50 data. These results confirm the result of previous research [79] by demonstrating that NEAT can evolve excellent policies in the mountain car task if the 20-input representation is used. In this case, structural mutations appear to have little effect on performance. This is surprising for two reasons. First, it suggests that one of NEAT's most powerful features, the ability to automatically optimize network topologies, is not helpful in the mountain car task. However, this result says less about the method than about the task, which is apparently simple enough to solve without complex topologies. Second, it demonstrates that NEAT can solve the mountain car task using exactly the same representation (SLPs with 20 inputs) on which Sarsa performs quite poorly. However, the two methods use these representations in different ways. Sarsa uses it to estimate a value function while NEAT uses it to estimate a policy in the form of an action selector. The latter may be simpler to represent since the outputs can have arbitrary value so long as the output corresponding to the best action has the highest value.

Given these results, we select the best performing configuration of each method (tile coding for Sarsa and the 20-input representation without structural mutations for NEAT) to conduct a careful comparison of their performance in the mountain car task. Specifically,

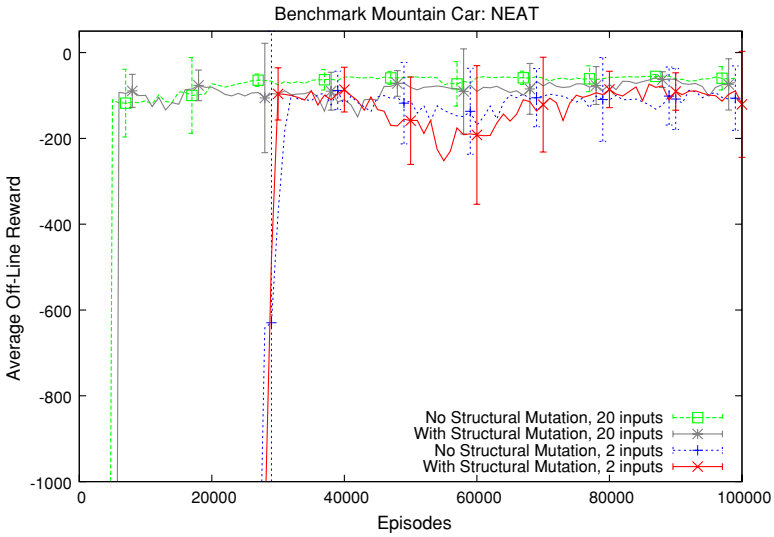


Fig. 7 A comparison of the average reward of the policies discovered by NEAT using various network representations in the benchmark mountain car task

we test each method for 50 independent runs, where each run lasts 100,000 episodes. Sarsa learners are tested with learning rates $\alpha = \{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$, exploration parameter settings of $\epsilon = \{0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$, and exploration decay settings of $d = \{0.99, 0.999, 1.0\}$, where the best performing parameters were found to be $\alpha = 0.1, \epsilon = 0.3$, and $d = 0.999$. NEAT was tested by setting the number of evaluations per organism to {1, 10, 50, 100}, and 50 was found superior.

In these experiments, as well as those reported later in this article, Sarsa and NEAT are not necessarily tested at the same number of parameter settings. Controlling for this factor is difficult, as different algorithms can have different numbers of parameters and those parameters can have different levels of sensitivity to performance. For example, while NEAT has many more parameters than Sarsa (see Table 2 in the Appendix), in our experience most of them have a negligible effect on performance. By contrast, setting Sarsa's few parameters well seems critical to successful learning. In each case, we use our intuition about each algorithm to select a range of parameters for testing that ensures it performs reasonably well. It is always possible that a more elaborate parameter search would further improve performance, though we think it is unlikely such improvements would cause qualitative changes in the results we present.

For each parameter setting, we estimate the performance at regular intervals of the best policy found so far by each method. For each run, these performance estimates are computed by averaging reward accrued over 1,000 test episodes. These results are then averaged across all 50 runs of each of the two methods for each given parameter setting. Figure 8 plots the results of these experiments, showing only the best performing parameter setting for each method. The final performance of both methods is quite similar and we believe it to be approximately optimal, as it matches the best results published by other researchers (e.g., [58,79]). At this scale, Sarsa appears to learn almost instantly; in fact, it requires on average about 3,000 episodes to find an approximately optimal policy. Additionally, for this task, the variance in the performance in NEAT is much higher than that of Sarsa. Although additional

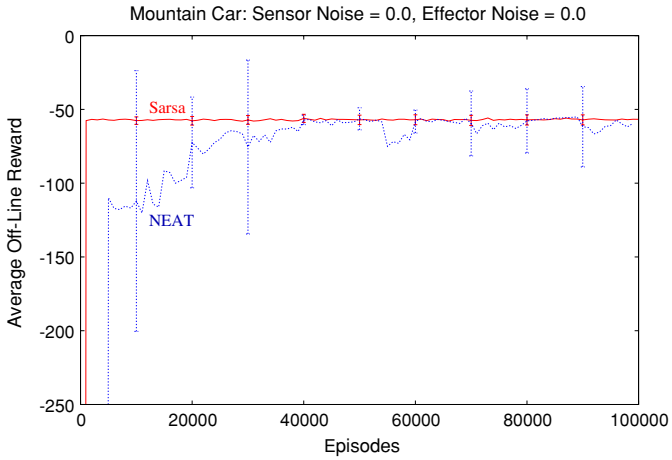


Fig. 8 A comparison of the average reward of the policies discovered by NEAT and Sarsa in the benchmark mountain car task

parameter tuning of NEAT may reduce this variance, the majority of results in this article show the same result; an experimenter who has reason to believe that the two methods will perform equally on a task *on average* may wish to select the method with the lower variance if there is only time for a single learning trial.

The most striking feature of these results is the great difference in speed between the two methods. While both methods eventually discover approximately optimal policies, NEAT requires orders of magnitude more episodes to do so. Student's t-tests confirm that the difference in performance between NEAT and Sarsa is statistically significant for the first 26,000 episodes ($p < 0.05$). The difference in learning speed is particularly striking considering that the tile coding representation is so much larger than the SLPs evolved by NEAT: the former has over 1,000 weights while the latter has only 60. Since mountain car is a fully observable task, the assumptions made by the Sarsa method (i.e., that the Markov property holds) are valid and thus these results lend empirical support to Sutton and Barto's [69] claim that TD methods, by exploiting the structure of the task, can be more efficient than policy search methods.

4.2 Keepaway

Since keepaway games are more computationally expensive, we conduct each run not for a fixed number of episodes, but until it plateaus, i.e., its performance does not improve for several simulator hours. Doing so enables us to generate more data with fixed computational resources. Since Sarsa runs plateau much sooner than NEAT runs (89 vs. 840 h[8] of simulator time, on average), we were able to conduct a total of 20 Sarsa runs and 5 NEAT runs. Sarsa players use previously established settings [63] of $\alpha = 0.05, \epsilon = 0.1$, and $d = 1.0$. NEAT uses the default parameter settings with structural mutations turned on [72] (see the "Appendix" for more details).

[8] For reference, 840 h of simulator time in the benchmark keepaway task corresponded to roughly 57 generations, 342,000 episodes, or 420 h of wall-clock time.

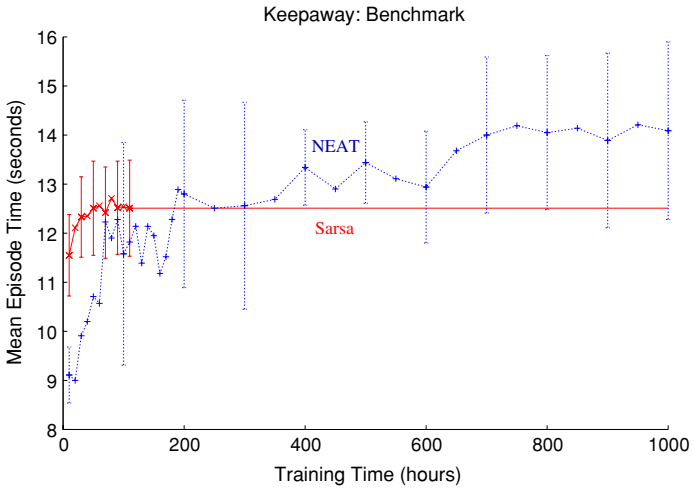


Fig. 9 A comparison of the average hold times of the policies discovered by NEAT and Sarsa in the benchmark keepaway task

As in mountain car, we estimate at regular intervals the performance of the best policy found so far by each method. For each run, these performance estimates are computed by averaging reward accrued over 1,000 test episodes. These results are then averaged across all runs of each of the two methods to obtain the plot shown in Fig. 9. Note that because the Sarsa learning curves plateau before the NEAT learning curves, the performance of the Sarsa learners is extended on the graph even after learning has finished, denoted by a horizontal performance line without plotted data points. For presentation purposes we plot the average performance every 10 h for the first 200 h and then every 50 h after that. Increasing the sampling resolution would not reveal any interesting detail in the learning curves.

As with mountain car, these results show a clear speed advantage for Sarsa: in the early part of learning its average policy is much better than NEAT's. However, unlike mountain car, these results also show that NEAT can learn substantially better policies in the long run. Student's t-tests confirm that the difference in performance between NEAT and Sarsa is statistically significant for times greater than or equal to 650 hours ($p < 0.014$). Since mountain car is a much smaller and simpler task than keepaway, the results obtained in it may suffer from a ceiling effect, i.e., NEAT cannot outperform Sarsa in the long run since both methods find near-optimal policies. By contrast, NEAT's slowness in keepaway is balanced by the quality of the best policies it ultimately discovers. Therefore, in more challenging domains there may be important trade-offs between speed and final performance.[9]

[9] This trade-off can occur only when Sarsa is combined with function approximation. In table-based systems, Sarsa is guaranteed to converge to the ϵ-optimal policy, so no policy search method could have substantially better asymptotic performance. The tasks we consider here require function approximation, whose performance can depend on the chosen representation. Therefore, we evaluate the methods using the best-performing representation for each method.

5 Testing the effect of sensor noise

In this section and the next, we use the results of the experiments conducted in the benchmark tasks to formulate hypotheses about the factors critical to the relative performance of Sarsa and NEAT. We then present variations of the benchmark tasks designed to test these hypotheses. This section explores why NEAT discovers better final policies in the benchmark keepaway task. We propose that this performance difference is due to the presence of noisy sensors in this task.

Specifically, we hypothesize that sensor noise reduces the final performance of Sarsa more than that of NEAT because sensor noise introduces a form of partial observability. Sarsa, like other TD methods, relies on an update rule that assumes a Markovian state representation, i.e., the state is defined such that the probability distribution over next states is independent of the agents' state and action histories. When the true state is only partially observable, Sarsa's convergence guarantees in the tabular cases no longer hold and learning may result in arbitrarily suboptimal policies or even catastrophic divergence of value function estimates. Sutton and Barto argue that TD methods can still be useful in many tasks that are not strictly Markov and conclude that "the inability to have access to a *perfect* Markov state representation is probably not a severe problem for a reinforcement learning agent" [69, Sect. 3.5]. While this claim is intuitive, it has not been rigorously tested in domains like mountain car or keepaway.

By contrast, NEAT and other policy search methods do not rely on the presence of the Markov property. Instead, they simply search for the best mapping from the given state representation to the available actions. If the sensors are noisy and that state representation is not Markov, even the best such mapping may be poor, since uncertainty about the state limits the agent's ability to determine what action to take. While this uncertainty imposes a ceiling on the performance of the resulting policy, NEAT can still search effectively up to that ceiling, as the divergence problems faced by TD methods do not occur.

We test our hypothesis that NEAT copes better with noisy sensors by devising variations of the benchmark tasks with various levels of sensor noise. The benchmark mountain car task is fully observable so we add different amounts of noise to the agent's sensors and then observe whether Sarsa or NEAT fares better as the Markov property fades away. In keepaway, the agents' sensors are already noisy. However, by eliminating this noise we can make the task effectively Markovian[10] and observe whether the relative performance of the two methods changes. The remainder of this section describes the results of experiments on these domain variations.

5.1 Partially observable mountain car

To make mountain car partially observable, we add Gaussian noise (with mean 0.0 and standard deviation σ) to the state features. For each experiment, Sarsa was optimized over settings of $\alpha = \{0.001, 0.005, 0.01, 0.02, 0.025, 0.05, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5\}$, $\epsilon = \{0.01, 0.05, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6\}$, and exploration decay $d = \{0.98, 0.99, 0.995, 0.999, 1.0\}$. All Sarsa results shown in this section use only the best performing parameter

[10] The state is still not truly Markovian because ball and player velocities are not included. If the agent stored past states it could calculate these velocities and therefore better predict future states. However, the keepaway benchmark task does not include velocities because past research did not find them useful for learning; players have low inertia and the field has a high coefficient of friction which means that velocities do not help agents learn in practice. In this paper we use the same state variables as previous work [63,64] and note that when sensor noise is removed the state is "effectively Markovian."

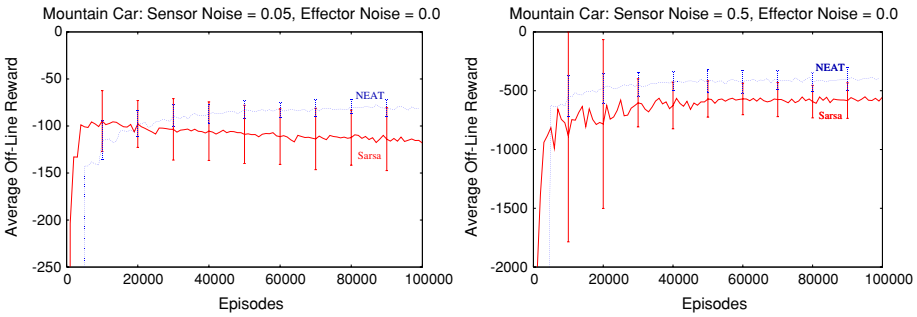


Fig. 10 A comparison of the average reward of the policies discovered by NEAT and Sarsa in the partially-observable variation of mountain car where $\sigma = 0.05$ (*left*) or $\sigma = 0.5$ (*right*)

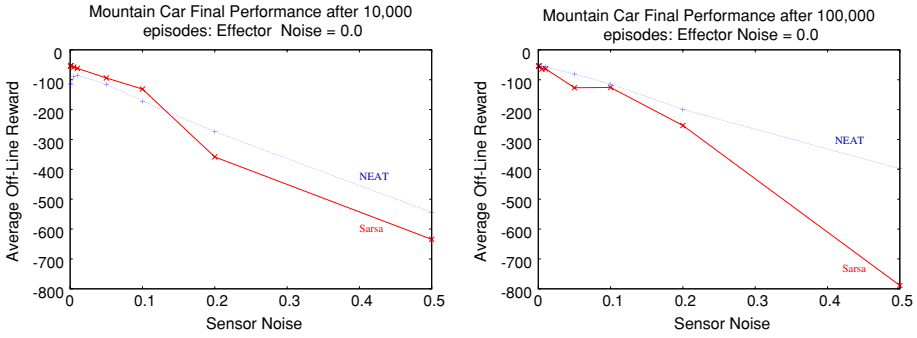


Fig. 11 A comparison of the average reward of the policies discovered by NEAT and Sarsa after 10,000 episodes (*left*) or 100,000 episodes (*right*) in mountain car variations with different levels of sensor noise

setting for the given value of σ. NEAT was tested with $\{10, 50, 100\}$ evaluations per organism, and 50 was again found superior for all experiments in this section.

The left side of Fig. 10 shows the relative performance of Sarsa and NEAT when $\sigma = 0.05$. As before, these results are averaged over 50 independent runs for each method. Not surprisingly, both methods perform worse than in the benchmark task since the agent is no longer certain about its state. Sarsa still learns much more rapidly than NEAT but now has substantially worse final performance. Student's t-tests confirm that the difference in performance between NEAT and Sarsa is statistically significant after 35,000 episodes ($p < 4.2 \times 10^{-5}$). Note that Sarsa's performance degrades slightly over time, which is not surprising since additional function approximation updates do not always lead to policy improvements. The learning parameters for Sarsa were tuned to maximize the final reward. They could also be tuned to minimize this 'unlearning' though the results are unlikely to be qualitatively different. The right side of Fig. 10 shows the relative performance of Sarsa and NEAT when $\sigma = 0.5$. Again Sarsa learns more quickly but NEAT has better final performance, though now the performance gap is even larger.

To verify that this trend is consistent, we also test other noise values where $0.05 < \sigma < 0.5$. Figure 11 summarizes the results of these experiments ("Appendix 2" details the learning parameters used). The left side shows performance early in learning (after 10,000 episodes) for each noise level and demonstrates that at low noise levels Sarsa retains its speed advantage over NEAT, though at higher noise levels NEAT outperforms it even early in learning.

The right side shows final performance (after 100,000 episodes) for each noise level. Because of the slight unlearning mentioned above, the final performance of Sarsa is not always its peak performance. However, using peak performance in this comparison instead would require violating the separation between training and testing. Recall that the points plotted in the graph come from multiple trials with a frozen policy. They are done purely for evaluation, and would not actually be performed by the learner in practice. Thus the learner would not know which policy used during learning was the best. In addition, since the amount of unlearning is small, the effect on the presented results is not substantial. These results demonstrate that NEAT consistently discovers better policies when the Markov property is removed from mountain car. Furthermore, the performance difference between them grows in direct proportion to the level of sensor noise. Hence, these experiments provide an initial confirmation of the hypothesis that sensor noise is more problematic for Sarsa than for NEAT.

5.2 Fully observable keepaway

Since the benchmark mountain car task is fully observable, we test whether the addition of sensor noise helps NEAT's relative performance. By contrast, the benchmark keepaway task is already partially observable, so we test whether the *removal* of sensor noise *hurts* NEAT's relative performance.

The computational expense of running keepaway episodes makes it prohibitive to test many intermediate noise values as we did with mountain car. However, in the case of no sensor noise, we conducted 5 runs of NEAT and 20 runs of Sarsa. (Initial results showed that the same learning settings as in the benchmark task were superior to other learning settings for both of the methods.) Figure 12 shows the results of these experiments. As in the benchmark version of the task (Fig. 9), Sarsa learns much more rapidly than NEAT. However, in the fully observable version, Sarsa also learns substantially better policies. Student's t-tests confirm that the difference in performance between NEAT and Sarsa is statistically significant for all points graphed ($p < 1.0 \times 10^{-4}$). These results provide additional confirmation of our hypothesis that full observability is a critical factor in Sarsa's performance. While Sarsa can learn well in the partially observable benchmark version of keepaway, its performance relative to NEAT improves dramatically when sensor noise is removed.

This outcome is surprising, since NEAT can evolve networks with as much or greater representational power than the RBFs used by Sarsa. Thus, the superior policies learned by Sarsa should, in principle, be discoverable by NEAT also. However, in practice, NEAT finds only local maxima in the space of network topologies and the space of weight settings for those topologies. In this case, those local maxima perform significantly worse than Sarsa.

Overall, these experiments about the effect of sensor noise in both mountain car and keepaway indicate that it can be a critical factor in the relative performance of evolutionary and TD methods. On one hand, the results confirm Sutton and Barto's claim that lack of a perfectly Markov state representation need not be a fatal problem for TD methods, since Sarsa continues to learn decent policies even at the highest noise levels tested. On the other hand, the loss of performance can be great enough to make Sarsa a less attractive choice than NEAT, as seen in the difference between the benchmark and fully-observable versions of keepaway. The results in mountain car show that Sarsa's performance degrades more rapidly as sensor noise increases, giving NEAT substantially better final performance. For lower levels of sensor noise, Sarsa can still be preferable if learning speed is more important than final performance. But the importance of the Markov property is underscored by the fact

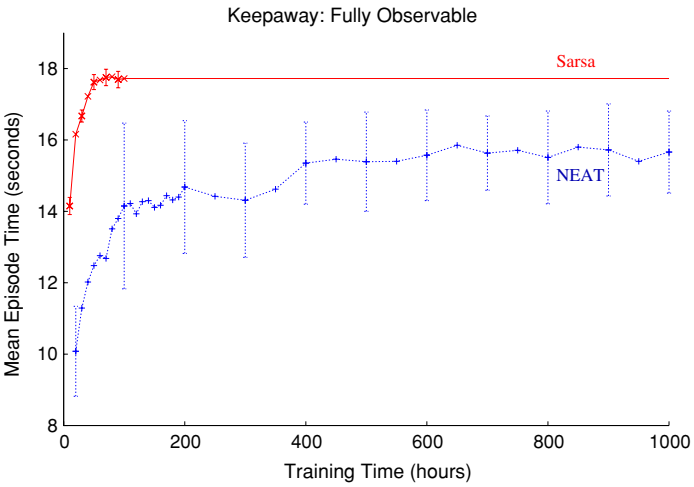


Fig. 12 A comparison of the average hold times of the policies discovered by NEAT and Sarsa in the fully-observable version of the keepaway task

that NEAT, which was dramatically slower in the benchmark task, actually learns faster than Sarsa at the higher noise levels.

6 Testing the effect of stochasticity

In this section, we look more closely at the differences in learning speed that occur in the benchmark tasks. We seek to identify the critical properties of these domains that explain why Sarsa initially learns a significantly better policy than NEAT.

We begin with the hypothesis that stochasticity of any kind, whether in the sensors, effectors, or initial state, reduces NEAT's learning speed more than Sarsa's. Recall that the fitness function used to evaluate each network consists of summing the reward obtained when using that network. Consequently, noise in the domain can render the fitness function unreliable, in which case resampling is crucial. If the required episodes per evaluation (EPE) increases as the domain becomes noisier, this could slow evolution down substantially, as the length of each generation grows in direct proportion to the EPE.[11]

By contrast, such stochasticity is unlikely to dramatically slow Sarsa's learning speed. The results presented in Sect. 5 demonstrate that sensor noise is problematic for Sarsa since it results in violations of the Markov property. However, the consequence is reduced final performance, not slower learning. Effector noise could potentially slow Sarsa by requiring a lower learning rate α. However, domains like mountain car and keepaway already require low learning rates for function approximation to be feasible, so this effect is likely to be negligible. Furthermore, effector noise could actually speed up learning by providing a nat-

[11] Increasing the EPE is an effective but not necessarily efficient way of increasing the accuracy of fitness estimates [11]. More sophisticated strategies that measure uncertainty when deciding which individuals to resample (e.g., [8,59]) may perform better. Studying such methods empirically is beyond the scope of this paper. However, even if they prove highly effective, they are likely to reduce but not eliminate the detrimental effect of resampling on the speed of evolutionary methods in noisy domains.

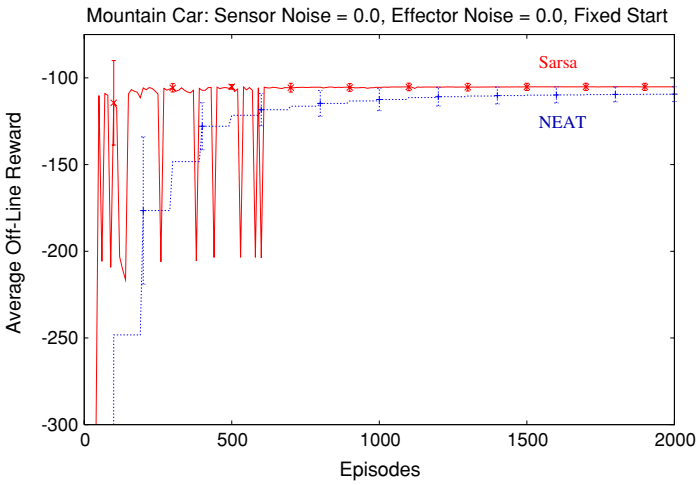


Fig. 13 A comparison of the average reward of the policies discovered by NEAT and Sarsa in the completely deterministic, i.e., fixed start state, version of mountain car. The *error bars* show the standard deviation

ural form of exploration. Stochasticity in the initial state is perhaps the least likely to slow learning, as it also provides natural exploration but does not require a lower α.

We test our hypothesis that stochasticity is more detrimental to NEAT's learning speed by devising variations of the benchmark tasks that are completely deterministic and then measuring whether NEAT's learning speed relative to Sarsa improves. Mountain car already lacks sensor and effector noise so rendering it deterministic requires only fixing the initial state. Making keepaway deterministic requires eliminating stochasticity in the sensors, effectors, and initial state. The remainder of this section describes the results of experiments on these deterministic variations.

6.1 Deterministic mountain car

In the benchmark version of mountain car, the transition function is deterministic, but the agent's initial state is random. Therefore, by fixing the agent's initial position at the bottom of the hill and the agent's initial velocity at zero ($x_t = \frac{\pi}{6}, \dot{x}_t = 0$), we obtain a completely deterministic variation of mountain car. Figure 13 shows the relative performance of Sarsa and NEAT in this task. As before, results are averaged over 50 independent runs for each method. Since no noise remains in the fitness function, NEAT uses an EPE of 1 rather than 50. Sarsa was tested on: $\alpha = \{0.001, 0.005, 0.01, 0.02, 0.025, 0.05, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5\}$, $\epsilon = \{0.01, 0.05, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6\}$, and exploration rate decay $d = \{0.98, 0.99, 0.995, 0.999, 1.0\}$. $\alpha = 0.3, \epsilon = 0.1, d = 0.995$ were found superior to other values. Note that both methods plateau at substantially lower values than in the benchmark task since the fixed state has lower than average value (the agent starts at the bottom of the hill with no momentum).

As in the benchmark task, both methods achieve approximately the same final performance but Sarsa achieves it more quickly. However, the difference in learning speed is greatly reduced. NEAT requires about 2,000 episodes to match Sarsa's performance, as opposed to 40,000 in the benchmark task. Therefore, these results provide initial confirmation of our hypothesis that stochasticity reduces the learning speed of NEAT more than that of Sarsa.

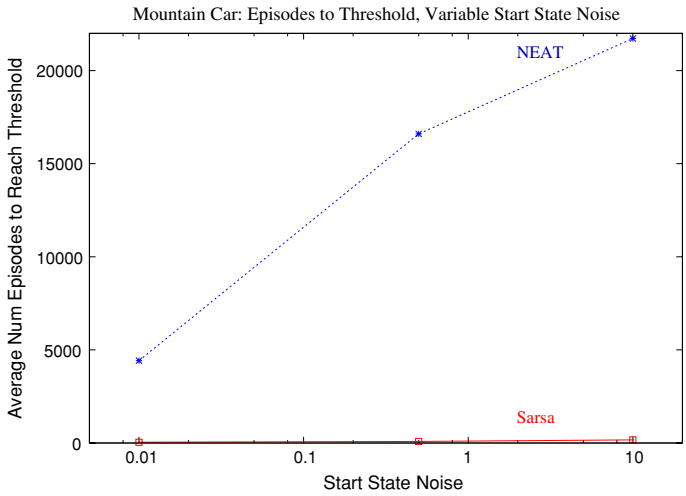


Fig. 14 A comparison of the average number of episodes required by NEAT and Sarsa to reach a near-optimal threshold in mountain car with different levels of noise in the start state

To verify that this trend is consistent, we also tested several intermediate cases, where the start state is neither deterministic nor completely random. Specifically, we studied cases where the agent's initial position and velocity are drawn from a Gaussian distribution with mean 0.0 and standard deviation $\sigma = \{0.01, 0.5, 10.0\}$. At each noise level, we tested Sarsa with learning rates $\alpha = \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$, exploration parameter settings of $\epsilon = \{0.01, 0.05, 0.1, 0.2, 0.3, 0.4\}$ and exploration decay settings of $d = \{0.885, 0.99, 0.995, 0.999, 1.0\}$. For the three settings of σ, we found $\alpha = \{0.2, 0.4, 0.2\}$, $\epsilon = \{0.1, 0.01, 0.01\}$, and $d = \{0.995, 0.995, 0.995\}$ to have superior performance, respectively. NEAT was tested with EPE$=\{1, 3, 5, 7, 10, 15, 20, 25, 30, 35, 40, 50, 55, 60, 65\}$ and was found to produce the highest performing policies when the EPE was set to $\{5, 25, 40\}$, respectively.

Figure 14 compares the number of episodes each method requires, at the best parameter setting, to reach a near-optimal threshold for different values of σ. We select thresholds that both methods are able to consistently reach before plateauing. For clarity, we show learning speed at only this near-optimal threshold, though the results are qualitatively similar for other thresholds too. Though the number of episodes Sarsa requires to meet the threshold appears relatively flat, it actually increases substantially, from 40 episodes when $\sigma = 0.01$ to 170 episodes when $\sigma = 10.0$. However, the magnitude of this change is dwarfed by that of NEAT, such that the difference in learning speed between the two methods increases dramatically. Thus, the results make clear that NEAT's disadvantage in terms of learning speed is highly correlated with the amount of noise in the fitness function. This is not surprising, since the best performing EPE increases in direct proportion to the noise. However, the Sarsa experiments do not have a similar trend. Overall, these results confirm our hypothesis that stochasticity in the fitness function is more detrimental to NEAT's learning speed.

However, it still possible for NEAT to perform better than Sarsa in early learning in a stochastic version of mountain car, despite its reduced learning speed. One example occurs in the results shown in Fig. 10. When sensor noise is increased, the time it takes for NEAT's performance to surpass Sarsa's actually decreases, even though the domain becomes more stochastic. The reason is that, in this case, all the additional stochasticity comes from sensor noise which, by introducing partial observability, greatly reduces the performance of Sarsa.

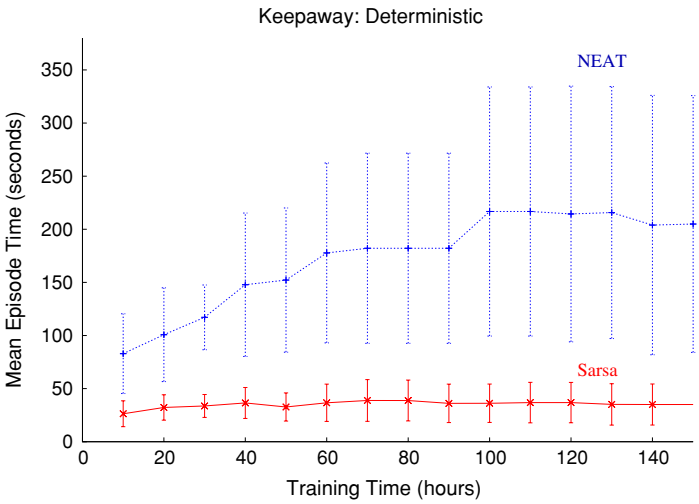


Fig. 15 A comparison of the average hold times of the policies discovered by NEAT and Sarsa in the deterministic version of the keepaway task

Thus, its performance in early learning relative to Sarsa actually improves in the noisier task. In this case, the additional sensor noise is not enough to change the necessary EPE (50 for both levels of sensor noise). Nonetheless, the results shown in Fig. 14 demonstrate that when stochasticity is increased enough to change the necessary EPE, NEAT's learning speed relative to Sarsa decreases quickly.

6.2 Deterministic keepaway

To further evaluate our hypothesis, we removed stochasticity in the sensors, effectors, and initial state of the keepaway task. We conducted 5 trials of NEAT and 20 trials of Sarsa in the resulting deterministic task. As with deterministic mountain car, NEAT spent only one episode evaluating each network. We hypothesized that Sarsa would benefit from an increased learning rate and tested several higher values, but found no improvement over the original value of 0.05. Figure 15 shows the results of these experiments, with mean hold times computed as before and averaged across all trials of each method. The difference in performance between NEAT and Sarsa is statistically significant for all points graphed ($p < 2.6 \times 10^{-6}$).

As in mountain car, removing stochasticity greatly improves NEAT's learning speed. In deterministic keepaway, the effect is even more striking, as NEAT actually learns faster than Sarsa, in addition to discovering dramatically superior policies. This outcome is unexpected since the deterministic version of the task is also fully observable and should be well suited to TD methods. The experiments suggest that, in the deterministic version of the task, the advantage Sarsa gains from full observability is far outweighed by the advantage NEAT gains from having rapid and accurate fitness evaluations.

The detrimental effects of noise on NEAT's learning speed are a direct result of the need for repeated resampling. Unlike TD methods, which can exploit the relationship between subsequent states to perform "bootstrapping" updates based on the Bellman equation, evolutionary methods treat the problem like a black box with an arbitrary fitness function to be maximized. Sutton and Barto [69, Sect. 1.3] argue that this is a serious weakness of evolutionary methods, which "ignore much of the useful structure of the reinforcement learning

Table 1 A summary of the results presented in Sects. 4, 5, and 6, showing, for each domain tested, which method has the best learning speed and final performance

Domain	Learning speed	Final performance
Benchmark mountain car	Sarsa	(tie)
Benchmark keepaway	Sarsa	NEAT
Partially observable mountain car	Sarsa	NEAT
Fully observable keepaway	Sarsa	Sarsa
Deterministic mountain car	Sarsa	(tie)
Deterministic keepaway	NEAT	NEAT

problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects."

It is no surprise then that evolutionary methods are so much slower when the fitness function is noisy enough to require substantial resampling. What is surprising is that they can perform so well when the fitness function is deterministic. In a small fully-observable task like mountain car, NEAT is still slower than Sarsa even in the deterministic version, though its speed improves by an order of magnitude. In keepaway, however, determinism allows NEAT to discover dramatically better policies and to do so faster than Sarsa.

Table 1 summarizes the results presented in Sects. 4, 5, and 6. It shows, for each domain tested, which method has the best learning speed and final performance. From this high-level perspective, it is clear that Sarsa's strength is its learning speed, as it proves faster than NEAT in all but one domain. By contrast, NEAT's strength is in final performance, as it ultimately discovers policies as good or better than Sarsa in all but one domain. The exceptions to these trends are remarkable because they further confirm our hypotheses about the critical factors involved. Sarsa is clearly inhibited by sensor noise, as the only task where it achieves a better final performance is a fully observable one. Similarly, NEAT is hindered by stochasticity, as the only task where it learns faster than Sarsa is a deterministic one.

Overall, these comparative results can help construct a broad outline of the strengths and weaknesses of the two methods tested, and possibly other evolutionary and TD methods too. Sarsa seems best suited to tasks that are fully observable, since it is designed to exploit the Markov property, or stochastic, since it is not particularly affected by noise in the transition function. NEAT seems best suited to tasks with noisy sensors, since it does not rely on the Markov property, or that are deterministic, since evolution is much more rapid when resampling is not required. In between these extremes, the picture is less clear and may depend on the task's particular eccentricities as well as the relative importance of learning speed and final performance.

7 Related work

The research presented in this article contributes to a small but growing body of empirical comparisons between TD and evolutionary methods. In this section, we survey previous research in this area.

By far the most popular benchmark domain for such comparisons is the pole balancing problem, also known as the inverted pendulum task. In this problem, the agent controls a

wheeled cart by applying forces to either side. The goal is to keep a pole attached to the cart upright, by constantly moving the cart in a way that maintains the pole's balance.

Whitley et al. [80] use the pole balancing task as a benchmark for testing GENITOR [81], a simple neuroevolutionary method that evolves the weights of a neural network with a manually designed topology. They compare its performance to Adaptive Heuristic Critic (AHC) [6], a TD method that uses neural networks to represent a value function and a control policy. They find that GENITOR is competitive with AHC's learning speed but more robust, in that a higher fraction of trials solve the task.

The version of the pole balancing task they consider has random start states but is otherwise deterministic, similar to the benchmark version of mountain car. Despite the noise in the fitness function, they find that increasing the EPE does not improve performance. They speculate this is because "the noise largely had a conservative effect: some good networks are lost because of poor start states, but it is more difficult for a poor net to obtain a good ranking" [80, p. 281]. In other words, a bad policy is not able to balance the pole for long, regardless of how favorable its initial state. This contrasts with mountain car, where certain initial states (e.g., those very near the goal and/or with high velocity) yield high scores even for policies that perform poorly in general. This qualitative difference may explain why increasing EPE helps in mountain car but not in pole balancing.

Moriarty and Miikkulainen [45] consider the same pole-balancing task and test the performance of Q-learning [76], the classic TD method, and Symbiotic, Adaptive Neuro-Evolution (SANE), which evolves population of neurons rather than entire networks. They find that SANE substantially outperforms Q-learning and an improved implementation of GENITOR on the pole balancing task with either random or fixed start states. AHC uses fewer episodes than SANE but substantially more computation time, which they attribute to the expense of performing backpropagation after each time step.

Several other researchers consider more challenging versions of the task, such as those with multiple poles or without velocity information, always using fixed start states [19,22,57, 60,77]. These studies compare the performance only of various evolutionary methods. However, Gomez et al. [21] recently conducted a comprehensive empirical study comparing the performance of several TD methods, including AHC, Sarsa, and Q-learning, to several evolutionary methods, including SANE, ESP, NEAT and Cooperative Synapse Neuroevolution (CoSyNE) [21]. They consider the original pole balancing task, as well as the double-pole variation with or without velocity information. Though certain TD methods occasionally outperform certain evolutionary methods, the latter fare much better overall, frequently solving the task one or more orders of magnitude more quickly.

Taken as a whole, these pole-balancing experiments make a strong case for the evolutionary approach to reinforcement learning. They show that such methods can find solutions even for the hardest versions of the task, sometimes much more quickly than TD methods. However, a critical caveat is the fact that most of the pole-balancing variations considered are completely deterministic. Even in those with random start states, the noise causes only the "conservative effect" noted by Whitley et al. such that one EPE remains sufficient for effective evolution. By contrast, the results presented in this article show that in truly noisy tasks, for which a higher EPE is essential, evolutionary methods can be much slower than the TD alternatives.

Of course, those same results also show that lack of speed can be balanced by better ultimate performance, as seen in keepaway. Unfortunately, none of the extensive pole-balancing results can confirm or disconfirm this effect, since they examine only CPU time and number of evaluations to reach a fixed performance threshold. They do not consider, as we do, the performance of each method once it plateaus.

Evolutionary and TD methods have been compared in other domains as well, mostly in competitive games. For example, Pollack and Blair [49] use a very simple self-play training paradigm (essentially an evolutionary method with a population size of two) to find neural network controllers to play backgammon. They compare their results to the performance of TD-Gammon [73], a neural network value function approximator that learned to play master-level backgammon using self-play. While Pollack and Blair's naïve method does not match TD-Gammon's performance, it does surprisingly well. They conclude that it is not TD methods that are central to Tesauro's success but rather the peculiar dynamics of backgammon that make self-play unusually effective. Tesauro [74] notes however that the performance gap between Pollack and Blair's approach and TD-Gammon is actually quite dramatic and that the former is probably unable to discover any nonlinear solutions.

Darwen [16] also applies neuroevolution to backgammon, using a full coevolutionary setup in which the members of each generation play a round-robin tournament against each other to obtain fitness estimates. In the linear setting, evolution ultimately outperforms TD, though it takes literally millions of games to do so. In the nonlinear setting, evolution never matches TD's performance. Darwen argues that finding good nonlinear solutions requires training on rare board positions and estimates that coevolution would require 50 million games to do so. By contrast, TD finds good nonlinear solutions in only 1.5 million games.

Similarly, Runarsson and Lucas [56] compare evolution and TD in small-board Go and find that TD learns much faster and in most cases achieves higher performance also. However, they find at least one setup, using coevolution, wherein evolution outperforms TD. They also present results for Othello [38], finding that TD methods are much faster but that a properly tuned evolutionary method ultimately performs best. Lucas and Togelius [39] present similar comparative results in a simple car-racing domain. They find that evolution achieves better fitness and is more reliable than TD but that, when successful, the latter learns faster.

In a set of three papers, Heidrich-Meisner and Igel compare a natural policy gradient method (a policy gradient method) with the covariance matrix adaptation evolution strategy (an evolutionary method) and find the evolutionary method to be generally superior. Each of these three papers compare performance on only a single simple task with a few settings: mountain car with and without observation noise for fixed and random start states [25], pole balancing with no noise and a random start state [24], and double pole balancing with no noise and a fixed start state [23]. This article differs not only in terms of methods compared, but also because we consider more settings (such as evaluating multiple levels of effector noise), perform tests on the significantly more complex task of keepaway, and form domain-independent conclusions about the two classes of methods considered.

All of these results contrast with those obtained in the pole-balancing domains in that they consistently find evolution to be much slower than TD methods. This discrepancy is not surprising given the qualitative differences between the domains. As noted above, most of the pole-balancing experiments use fixed start states and hence completely deterministic fitness functions. Even in those cases with random start states, the "conservative effect" of noise in pole balancing means 1 EPE remains sufficient. Consequently, evolution is able to progress as fast or faster than TD methods. By contrast, the results showing evolution to be slower than TD use domains with very noisy fitness functions. Competitive games have this property naturally, since fitness depends greatly on the particular opponent. In the car-racing domain, the racetrack is selected randomly at the beginning of each episode.

Hence, these results are broadly consistent with a main finding of this article: stochasticity is more detrimental to evolution than to TD methods. However, none of these previous results directly examine this factor in a controlled way. They do not, as we do in Sect. 6, compare

the effect of varying levels of stochasticity in the *same* domain. Therefore, we believe our results provide more rigorous evidence in support of this conclusion.

Some previous work has also considered the effect of partial observability on the relative performance of evolutionary and TD methods. Moriarty et al. [46] survey evolutionary methods for reinforcement learning and note that they should be less vulnerable to the consequences of incomplete state information since they do not directly rely on the Markov property. They verify this claim using a simple 4-state POMDP, wherein an evolutionary method substantially outperforms Q-learning. This article presents additional evidence in support of this conclusion, with results from more realistic problems with larger, continuous state spaces. As with stochasticity, we also study the effect of varying levels of partial observability.

Several researchers (e.g., [19,60]), have noted that neuroevolution is particularly well-suited to partially observable tasks, not only because it does not rely on the Markov property, but because it can evolve networks with recurrent connections. These connections serve as a form of memory that can help reduce ambiguity in the agent's observations. Several results in the pole-balancing domain confirm this hypothesis, as the variations without velocity information cannot be solved without recurrency.[12] Gomez and Schmidhuber [20] present a neuroevolutionary method specifically designed to tackle "deep-memory POMDPs" in which the agent must remember observations from many time steps ago. They show that it outperforms a TD approach using Long Short-Term Memory (LSTM) [4] to cope with partial observability.

Metzen et al. [43] also investigate learning in the keepaway domain. They use Evolutionary Acquisition of Neural Topologies (EANT) [29], a neuroevolutionary method similar to NEAT. They show that EANT's performance improves when trained in fully observable keepaway, relative to the benchmark task. However, they do not study a deterministic version or directly compare their results with other learning methods but focus instead on better understanding EANT through ablation studies.

Finally, Kalyanakrishnan and Stone [28] compare Sarsa and the cross-entropy method [41,71], another approach to policy search, in a simple navigation task. They study how the relative performance of these methods changes with respect to several domain characteristics, including sensor and effector noise. As in this article, they conclude that sensor noise reduces the final performance of Sarsa more than that of the policy search alternative. They do not observe, as we do, that stochasticity reduces the learning speed of policy search more than Sarsa. However, in their experiments, the cross entropy method uses a large, fixed EPE. Hence, they do not examine whether policy search can be sped up in less stochastic domains by reducing resampling. They also compare the two methods in the benchmark keepaway task and find that the cross-entropy method, like NEAT, is slower than Sarsa. However, its final performance is only as good as Sarsa, i.e., not as good as NEAT. The fact that the cross-entropy method was used to evolve the weights of a fixed-topology neural network may account its lower final performance compared to NEAT, which can evolve topologies customized to the task.

[12] Informal experiments in the benchmark version of keepaway and the stochastic version of mountain car suggest that enabling recurrent links does not increase final performance but does slow learning. Thus, in all of our experiments NEAT evolves only networks without recurrent links.

8 Future work

Much empirical work remains to be done to obtain a thorough understanding of the relative merits of TD and evolutionary methods for reinforcement learning. The work presented here could be extended along two dimensions: methods and domains.

NEAT and Sarsa are important representative methods, but their performance by no means tells the whole story. Other evolutionary methods such as CoSyNE [21], EANT [29], and HyperNEAT [17], an extension to NEAT based on indirect encodings, also deserve closer empirical study. Beyond evolutionary methods, other policy search approaches such as the cross-entropy method [41,71] or policy gradient approaches [3,7,34,70] could be usefully compared with TD methods. Similarly, recent developments in making value function approximation more robust, e.g., least-squares policy iteration [36], fitted Q-iteration [54] and evolutionary function approximation [79], need to be thoroughly compared to the traditional function approximation approach used in this paper. In addition, TD methods for automatically constructing basis functions [31,40,42] are excellent candidates for comparison with NEAT's capacity for finding good representations. Other value function approaches, such as model-based methods [14,30,65], could also be compared to evolutionary and other policy search methods.

A broader range of benchmark domains is also necessary to improve our understanding of when each approach excels. In the last few years, reinforcement learning methods have been successfully applied to several challenging, realistic tasks, such as elevator control [15], helicopter control [47], the game of Tetris [71], autonomic resource allocation [75], and server job scheduling [79]. Such tasks could serve as excellent testbeds for comparisons between disparate methods.

Finally, the results using NEAT without structural mutations in the benchmark mountain car task (see Sect. 4.1) suggests that further study about the role of topology evolution in NEAT's success would be useful. While some ablation experiments in the double pole-balancing task have shown that structural mutations can be essential to NEAT's success [60], our results confirm the intuition that, in some tasks, evolving structure is either unnecessary or too difficult to be helpful. Additional experiments in the mountain car domain using multiple structural mutation rates would help establish conclusively if it is possible to outperform simple weight evolution in SLPs. Furthermore, ablation experiments in keepaway could help determine the importance of topology evolution in that more complex task, since in this article we evaluate NEAT only with default parameter settings. Such experiments are unlikely to qualitatively alter the results presented here, but they could contribute to ongoing research about what domain characteristics affect the feasibility of topology-evolving neuroevolution [32,33].

9 Conclusion

This article presents results of empirical comparisons between Sarsa and NEAT in mountain car and keepaway and tests two specific hypotheses about the critical factors contributing to these methods' relative performance: (1) that sensor noise reduces the final performance of Sarsa more than that of NEAT, because Sarsa's learning updates are not reliable in the absence of the Markov property, and (2) that stochasticity, by introducing noise in fitness estimates, reduces the learning speed of NEAT more than that of Sarsa. Experiments in variations of mountain car and keepaway designed to isolate these factors confirm both these hypotheses.

This article's experiments contribute to a body of empirical comparisons between TD and evolutionary methods that is much in need of expansion. In addition to formulating and testing specific hypotheses with these methods in concrete domains, we hope that this article will encourage further empirical comparisons of a similar nature that are needed to achieve a complete picture of the relative strengths of TD and evolutionary methods for reinforcement learning.

Acknowledgments We would like to thank Ken Stanley for help setting up NEAT in keepaway, as well as Shivaram Kalyanakrishnan, Nate Kohl, Frans Oliehoek, David Pardoe, Jefferson Provost, Joseph Reisinger, Ken Stanley, and the anonymous reviewers for helpful comments and suggestions. This research was supported in part by NSF CAREER award IIS-0237699 and NSF award EIA-0303609.

Appendix 1: NEAT parameters

Tables 2 and 3 detail the NEAT parameters used in our experiments. These settings were not varied but were taken from the default settings included with the NEAT package. Note that the default settings in the two experiments differ slightly because they rely on two separate versions of NEAT (the C++ version is used for mountain car and the C version is used for keepaway). Stanley and Miikkulainen [60] describe the semantics of these parameters in detail.

Table 2 The NEAT parameters in this table are used in all mountain car experiments

Parameter	Value	Parameter	Value	Parameter	Value
Weight-mut-power	0.005	Recur-prop	0.0	Disjoint-coeff (c_1)	1.0
Excess-coeff (c_2)	1.0	Mutdiff-coeff (c_3)	0.4	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)	0.0	Mutate-add-link-prob (m_l)	0.0
Interspecies-mate-rate	0.001	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	10×10^6	Newlink-tries	20
Babies-stolen	0	Num-compat-mod	0.3	Num-species-target	4

Appendix 2: Supplemental experiment parameters

Figure 6 compares learning in the benchmark mountain car task with five different function approximators. The tile coding formulation performs significantly better than the others, thus its parameter settings are detailed in Sect. 4.1. The settings of the other four function approximators are detailed here. Output and hidden nodes were sigmoidal $\left(\frac{1.0}{1.0+e^{-x}}\right)$. In all cases,

Table 3 The NEAT parameters in this table are used in all keepaway experiments

Parameter	Value	Parameter	Value	Parameter	Value
Weight-mut-power	2.5	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	0.05	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	1,000	Newlink-tries	20
Babies-stolen	0	Num-compat-mod	0.3	Num-species-target	20

parameters in the tests were tuned from the following sets: $\alpha = \{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$, $\epsilon = \{0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$, and $d = \{0.990, 0.999, 1.0\}$.

- *SLP: 2 inputs*: The single layer perceptron has two real valued inputs, one for the car's velocity and one for the car's position. We found $\alpha = 0.001$, $\epsilon = 0.1$,, and $d = 0.999$ superior.
- *SLP: 20 inputs*: This single layer perceptron has 20 binary valued inputs, 10 for the car's velocity and 10 for the car's position. We found $\alpha = 0.001$, $\epsilon = 0.1$, and $d = 0.990$ superior.
- *MLP: 2 inputs*: The multi-layer perceptron has two real valued inputs, one for the car's velocity and one for the car's position, and two hidden nodes. We found $\alpha = 0.3$, $\epsilon = 0.1$, and $d = 0.999$ superior.
- *MLP: 20 inputs*: This multi-layer perceptron has 20 binary valued inputs, 10 for the car's velocity and 10 for the car's position, and two hidden nodes. We found $\alpha = 0.001$, $\epsilon = 0.01$, and $d = 0.990$ superior.

Note that both MLP configurations use two hidden nodes. This choice, which was not tuned, was based on our intuition. However, when NEAT is run on the benchmark mountain car problem, after training with structural mutations enabled (as done in Fig. 7), it learns topologies that have two hidden nodes, on average. Specifically, over 50 independent trials, we found the final champion topologies to have an average of 2.3 hidden nodes, with a standard deviation of 0.51. The remainder of mountain car NEAT experiments have structural mutation disabled, analogous to the SLP topologies.

Figure 7 investigates the effect of using different numbers of inputs and structural mutation on mountain car policy performance. The two *No Structural Mutation* experiments set the mutate-add-node-prob (m_n) parameter and the mutate-add-link-prob (m_l) parameter to zero. The two *With Structural Mutation* experiments set these two parameters to their default values ($m_n = 0.02, m_l = 0.1$). As above, the 20 input experiments used twenty binary inputs and the 2 input experiments used 2 real-valued inputs. All experiments used an EPE of 50.

Figure 10 summarizes a set of experiments detailing how policy performance changes with sensor noise (effector noise is zero throughout). All NEAT experiments used 50 EPE. Sarsa settings are as follows:

- $\sigma = 0.0 : \ \alpha = 0.1, \ \epsilon = 0.1, \ d = 0.990$
- $\sigma = 0.0005 : \ \alpha = 0.1, \ \epsilon = 0.3, \ d = 0.990$

- $\sigma = 0.001 :\ \alpha = 0.1,\ \epsilon = 0.2,\ d = 0.990$
- $\sigma = 0.005 :\ \alpha = 0.1,\ \epsilon = 0.3,\ d = 0.999$
- $\sigma = 0.01 :\ \alpha = 0.1,\ \epsilon = 0.3,\ d = 0.999$
- $\sigma = 0.05 :\ \alpha = 0.1,\ \epsilon = 0.2,\ d = 0.990$
- $\sigma = 0.1 :\ \alpha = 0.001,\ \epsilon = 0.1,\ d = 0.999$
- $\sigma = 0.2 :\ \alpha = 0.001,\ \epsilon = 0.1,\ d = 0.999$
- $\sigma = 0.5 :\ \alpha = 0.01,\ \epsilon = 0.1,\ d = 0.990$

References

1. Albus, J. S. (1981). *Brains, behavior, and robotics*. Peterborough, NH: Byte Books.
2. Anderson, C. W. (1986). *Learning and problem solving with multilayer connectionist systems*. Ph.D. thesis, University of Massachusetts, Amherst, MA.
3. Baird, L., & Moore, A. (1999). Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems* (Vol. 11). Cambridge, MA: MIT Press.
4. Bakker, B. (2002). Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems* (Vol. 14, pp. 1475–1482).
5. Barto, A., & Duff, M. (1994). Monte Carlo matrix inversion and reinforcement learning. In *Advances in Neural Information Processing Systems* (Vol. 6, pp. 687–694).
6. Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-13*(5), 834–846.
7. Baxter, J., & Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research, 15*, 319–350.
8. Beielstein, T., & Markon, S. (2002). Threshold selection, hypothesis tests and DOE methods. *2002 Congresss on evolutionary computation* (pp. 777–782).
9. Bellman, R. E. (1956). A problem in the sequential design of experiments. *Sankhya, 16*, 221–229.
10. Bellman, R. E. (1957). *Dynamic programming*. Princeton: Princeton University Press.
11. Beyer, H.-G., & Sendhoff, B. (2007). Evolutionary algorithms in the presence of noise: To sample or not to sample. In *Proceedings of the 1st IEEE Symposium on Foundations of Computational Intelligence* (pp. 17–24).
12. Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems* (Vol. 7).
13. Bradtke, S. J., & Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In *Advances in Neural Information Processing Systems* (Vol. 7, pp. 393–400).
14. Brafman, R. I., & Tennenholtz, M. (2002). R-MAX—a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research, 3*, 213–231.
15. Crites, R. H., & Barto, A. G. (1998). Elevator group control using multiple reinforcement learning agents. *Machine Learning, 33*(2-3), 235–262.
16. Darwen, P. J. (2001). Why co-evolution beats temporal difference learning at backgammon for a linear architecture, but not a non-linear architecture. In *Proceedings of the 2001 Congress on Evolutionary Computation* (pp. 1003–1010).
17. Gauci, J. J., & Stanley, K. O. (2007). Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Genetic and Evolutionary Computation Conference*.
18. Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Boston, MA: Addison-Wesley.
19. Gomez, F., & Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 1356–1361).
20. Gomez, F., & Schmidhuber, J. (2005). Co-evolving recurrent neurons learn deep memory pomdps. In *GECCO-05: Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 491–498).
21. Gomez, F., Schmidhuber, J., & Miikkulainen, R. (2006). Efficient non-linear control through neuro-evolution. In *Proceedings of the European Conference on Machine Learning*.
22. Gruau, F., Whitley, D., & Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In *Genetic Programming 1996: Proceedings of the 1st Annual Conference* (pp. 81–89).

23. Heidrich-Meisner, V., & Igel, C. (2008a). Evolution strategies for direct policy search. In *Proceedings of the 10th International Conference on Parallel Problem Solving from Nature* (pp. 428–437). Berlin, Heidelberg: Springer.
24. Heidrich-Meisner, V., & Igel, C. (2008b). Similarities and differences between policy gradient methods and evolution strategies. In *Proceedings of the 16th European Symposium on Artificial Neural Networks (ESANN)*.
25. Heidrich-Meisner, V., & Igel, C. (2008c). Variable metric reinforcement learning methods applied to the noisy mountain car problem. In *Recent Advances in Reinforcement Learning: 8th European Workshop* (pp. 136–150). Berlin, Heidelberg: Springer.
26. Jong, N. K., & Stone, P. (2007). Model-based exploration in continuous state spaces. In *The 7th Symposium on Abstraction, Reformulation, and Approximation*.
27. Kakade, S. (2003). *On the sample complexity of reinforcement learning*. Ph.D. thesis, University College London, London, UK.
28. Kalyanakrishnan, S., & Stone, P. (2009). An empirical analysis of value function-based and policy search reinforcement learning. In *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2009)*.
29. Kassahun, Y., & Sommer, G. (2005). Automatic neural robot controller design using evolutionary acquisition of neural topologies. In *Fachgespräch Autonome Mobile Systeme (AMS 2005), Stuttgart, Germany, 8, 9.12.05*, Informatik aktuell (Vol. 19, pp. 315–321). Springer.
30. Kearns, M., & Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning, 49*(2), 209–232.
31. Keller, P., Mannor, S., & Precup, D.(2006). Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning* (pp. 449–456).
32. Kohl, N., & Miikkulainen, R. (2008). Evolving neural networks for fractured domains. In *Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 1405–1412).
33. Kohl, N., & Miikkulainen, R. (2009). Evolving neural networks for strategic decision-making problems. *Neural Networks, Special Issue on Goal-Directed Neural Systems*, *22*(3), 326–337.
34. Kohl, M., & Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation* (pp. 2619–2624).
35. Kretchmar, R. M., & Anderson, C. W. (1997). Comparison of CMACs and radial basis functions for local function approximators in reinforcement learning. In *International Conference on Neural Networks*.
36. Lagoudakis, M. G., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research, 4*, 1107–1149.
37. Littman, M. L., Dean, T. L., & Kaelbling, L. P. (1995). On the complexity of solving Markov decision processes. In *Proceedings of the 11th International Conference on Uncertainty in Artificial Intelligence* (pp. 394–402).
38. Lucas, S. M., & Runarsson, T. P. (2006). Temporal difference learning versus co-evolution for acquiring Othello position evaluation. In *IEEE Symposium on Computational Intelligence and Games*.
39. Lucas, S. M., & Togelius, J. (2007). Point-to-point car racing: An initial study of evolution versus temporal difference learning. In *IEEE Symposium on Computational Intelligence and Games* (pp. 260–267).
40. Mahadevan, S. (2005). Samuel meets Amarel: Automating value function approximation using global state space analysis. In *Proceedings of the 20th National Conference on Artificial Intelligence*.
41. Mannor, S., Rubenstein, R., & Gat, Y. (2003). The cross-entropy method for fast policy search. In *Proceedings of the 20th International Conference on Machine Learning* (pp. 512–519).
42. Menache, I., Mannor, S., & Shimkin, N. (2005). Basis function adaptation in temporal difference reinforcement earning. *Annals of Operations Research, 134*, 215–238.
43. Metzen, J. H., Edgington, M., Kassahun, Y., & Kirchner, F. (2008). Analysis of an evolutionary reinforcement learning method in a multiagent domain. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2008)* (pp. 291–298). Estoril, Portugal.
44. Moore, A., & Atkeson, C. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning, 13*, 103–130.
45. Moriarty, D. E., & Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning, 22*(11), 11–33.
46. Moriarty, D. E., Schultz, A. C., & Grefenstette, J. J. (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research, 11*, 99–229.

47. Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., et al. (2004). Inverted autonomous helicopter flight via reinforcement learning. In *Proceedings of the International Symposium on Experimental Robotics*.
48. Noda, I., Matsubara, H., Hiraki, K., & Frank, I. (1998). Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence, 12*, 233–250.
49. Pollack, J., & Blair, A. (1998). Co-evolution in the successful learning of backgammon strategy. *Machine Learning, 32*, 225–240.
50. Potter, M. A., & Jong, K. A. D. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation, 8*, 1–29.
51. Powell, M. (1987). *Radial basis functions for multivariate interpolation: A review algorithms for approximation*. Oxford: Clarendon Press.
52. Pyeatt, L. D., & Howe, A. E. (2001). Decision tree function approximation in reinforcement learning. In *Proceedings of the 3rd International Symposium on Adaptive Systems: Evolutionary computation and probabilistic graphical models* (pp. 70–77).
53. Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimization. *Neural Computing and Applications, 1*(1), 67–90.
54. Reidmiller, M. (2005). Neural fitted Q iteration—first experiences with a data efficient neural reinforcement learning method. In *Proceedings of the 16th European Conference on Machine Learning* (pp. 317–328).
55. Rummery, G., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems CUED/F-INFENG/TR 166*. Cambridge University.
56. Runarsson, T. P., & Lucas, S. M. (2005). Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go. *IEEE Transactions on Evolutionary Computation, 9*, 628–640.
57. Saravanan, N., & Fogel, D. B. (1995). Evolving neural control systems. *IEEE Expert: Intelligent Systems and Their Applications, 10*(3), 23–27.
58. Smart, W. D., & Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *Proceedings of the 17th International Conference on Machine Learning* (pp. 903–910).
59. Stagge, P. (1998). Averaging efficiently in the presence of noise. *Parallel Problem Solving from Nature, 5*, 188–197.
60. Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation, 10*(2), 99–127.
61. Stanley, K. O., & Miikkulainen, R. (2004). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research, 21*, 63–100.
62. Stone, P. (2000). *Layered learning in multiagent systems: A winning approach to robotic soccer*. Cambridge, MA: MIT Press.
63. Stone, P., Kuhlmann, G., Taylor, M. E., & Liu, Y. (2005a). Keepaway soccer: From machine learning testbed to benchmark. In *RoboCup-2005: Robot Soccer World Cup IX* (Vol. 4020, pp. 93–105). Berlin: Springer.
64. Stone, P., Sutton, R. S., & Kuhlmann, G. (2005). Learning in RoboCup-soccer keepaway. *Adaptive Behavior, 13*(3), 165–188.
65. Strehl, A., & Littman, M. (2005). A theoretical analysis of model-based interval estimation. In *Proceedings of the 22nd International Conference on Machine Learning* (pp. 856–863).
66. Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems* (Vol. 8, pp. 1038–1044).
67. Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning, 3*, 9–44.
68. Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning* (pp. 216–224).
69. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.
70. Sutton, R., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems* (pp. 1057–1063).
71. Szita, I., & Lörincz, A. (2006). Learning Tetris using the noisy cross-entropy method. *Neural Computation, 18*(12), 2936–2941.
72. Taylor, M. E., Whiteson, S., & Stone, P. (2006). Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *GECCO 2006: Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 1321–1328).

73. Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program achieves master-level play. *Neural Computation, 6*, 215–219.
74. Tesauro, G. (1998). Comments on "co-evolution in the successful learning of backgammon strategy". *Machine Learning, 32*(3), 241–243.
75. Tesauro, G., Das, N. K. J. R., & Bennania, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 3rd International Conference on Autonomic Computing*.
76. Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning, 8*(3-4), 9–44.
77. Weiland, A. (1991). Evolving neural network controllers for unstable systems. In *International Joint Conference on Neural Networks* (pp. 667–673).
78. Whiteson, S., Kohl, N., Miikkulainen, R., & Stone, P. (2005). Evolving keepaway soccer players through task decomposition. *Machine Learning, 59*(1), 5–30.
79. Whiteson, S., & Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research, 7*, 877–917.
80. Whitley, D., Dominic, S., Das, R., & Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning, 13*, 259–284.
81. Whitley, D., & Kauth, K. (1988). GENITOR: A different genetic algorithm. In *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence* (pp. 118–130).
82. Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE, 87*(9), 1423–1447.