From Reinforcement Learning to Deep Reinforcement Learning: An Overview

Forest Agostinelli, Guillaume Hocquet, Sameer Singh, and Pierre Baldi(✉)

University of California - Irvine, Irvine, CA 92697, USA
{fagostin,sameer,pfbaldi}@uci.edu

Abstract. This article provides a brief overview of reinforcement learning, from its origins to current research trends, including deep reinforcement learning, with an emphasis on first principles.

Keywords: Machine learning · Reinforcement learning
Deep learning · Deep reinforcement learning

1 Introduction

This article provides a concise overview of reinforcement learning, from its origins to deep reinforcement learning. Thousands of articles have been written on reinforcement learning and we could not cite, let alone survey, all of them. Rather we have tried to focus here on first principles and algorithmic aspects, trying to organize a body of known algorithms in a logical way. A fairly comprehensive introduction to reinforcement learning is provided by [113]. Earlier surveys of the literature can be found in [33,46,51].

1.1 Brief History

The concept of reinforcement learning has emerged historically from the combination of two currents of research: (1) the study of the behavior of animals in response to stimuli; and (2) the development of efficient approaches to problems of optimal control.

In behavioral psychology, the term *reinforcement* was introduced by Pavlov in the early 1900s, while investigating the psychology and psychopathology of animals in the context of conditioning stimuli and conditioned responses [47]. One of his experiments consisted in ringing a bell just before giving food to a dog; after a few repetitions, Pavlov noticed that the sound of the bell alone made the dog salivate. In classical conditioning terminology, the bell is the previously neutral stimulus, which becomes a *conditioned stimulus* after becoming associated with the *unconditioned stimulus* (the food). The conditioned stimulus eventually comes to trigger a conditioned response (salivation). Conditioning

G. Hocquet—Work performed while visiting the University of California, Irvine.

L. Rozonoer et al. (Eds.): Braverman Readings in Machine Learning, LNAI 11100, pp. 298–328, 2018.
https://doi.org/10.1007/978-3-319-99492-5_13

experiments led to Thorndike's Law of Effect [118] in 1911, which states that: "Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur".

This formed the basis of *operant conditioning* (or instrumental conditioning) in which: (1) the strength of a behavior is modified by the behavior's consequences, such as reward or punishment; and (2) the behavior is controlled by antecedents called "discriminative stimuli" which come to emit those responses. Operant conditioning was studied in the 1930s by Skinner, with his experiments on the behavior of rats exposed to different types of reinforcers (stimuli).

A few years later, in the Organization of Behavior [39] (1949), Hebb proposed one of the first theories about the neural basis of learning using the notions of cell assemblies and "Hebbian" learning, encapsulated in the sentence "When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." These are some of the biological underpinnings and sources of inspiration for many subsequent developments in reinforcement learning and other forms of learning, such as supervised learning.

In 1954, in the context of optimal control theory, Bellman introduced dynamic programming [9], and the concept of value functions. These functions are computed using a recursive relationship, now called the Bellman equation. Bellman's work was within the framework of Markov Decision Process (MDPs), which were studied in detail by [44]. One of Howard's students, Drake, proposed an extension with partial observability: the POMDP models [27].

In 1961, [70] discussed several issues in the nascent field of reinforcement learning, in particular the problem of *credit assignment*, which is one of the core problems in the field. Around the same period, reinforcement learning ideas began to be applied to games. For instance, Samuel developed his checkers player [93] using Temporal Differences method. Other experiments were carried by Michie, including the development of the MENACE system to learn how to play Noughts and Crosses [67,68], and the BOXES controller [69] which has been applied to pole balancing problems.

In the 1970s, Tsetlin made several contributions within the area of Automata, in particular in relation to the n-armed bandit problem, i.e. how to select which levers to pull in order to maximize the gain in a game comprising n slot machines without initial knowledge. This problem can be viewed as a special case of a reinforcement learning problem with a single state. In 1975, Holland developed genetic algorithms [42], paving the way for reinforcement learning based on evolutionary algorithms.

In 1988, [126] presented the REINFORCE algorithms, which led to a variety of policy gradient methods. The same year, Sutton introduced TD(λ) [111]. In 1989, Watkins proposed the Q-Learning algorithm [123].

1.2 Applications

Reinforcement learning methods have been effective in a variety of areas, in particular in games. Success stories include the application of reinforcement learning to stochastic games (Backgammon [117]), learning by self-play (Chess [56]), learning from games played by experts (Go [100]), and learning without using any hand-crafted features (Atari games [72]).

When the objective is defined by a control task, reinforcement learning has been used to perform low-speed sustained inverted hovering with an helicopter [77], balance a pendulum without a priori knowledge of its dynamics [3], or balance and ride a bicycle [88]. Reinforcement learning has also found plenty of applications in robotics [52], including recent success in manipulation [59] and locomotion [97]. Other notable successes include solutions to the problems of elevator dispatching [19], dynamic communication allocation for cellular radio channels [104], job-shop scheduling [129], and traveling salesman optimization [26]. Other potential industrial applications have included packet routing [12], financial trading [73], and dialog systems [58].

1.3 General Idea Behind Reinforcement Learning

Reinforcement learning is used to compute a behavior strategy, a *policy*, that maximizes a satisfaction criteria, a long term sum of *rewards*, by interacting through *trials and errors* with a given environment (Fig. 1).

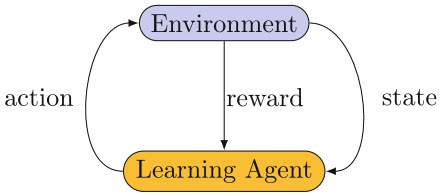

Fig. 1. The agent-environment interaction protocol

A reinforcement learning problem consists of a decision-maker, called the *agent*, operating in an *environment* modeled by *states* $s_t \in \mathcal{S}$. The agent is capable of taking certain *actions* $a_t \in \mathcal{A}(s_t)$, as a function of the current state s_t. After choosing an action at time t, the agent receives a scalar *reward* $r_{t+1} \in \mathbb{R}$ and finds itself in a new state s_{t+1} that depends on the current state and the chosen action.

At each time step, the agent follows a strategy, called the *policy* π_t, which is a mapping from states to the probability of selecting each possible action: $\pi(s, a)$ denotes the probability that $a = a_t$ if $s = s_t$.

The objective of reinforcement learning is to use the interactions of the agent with its environment to derive (or approximate) an optimal policy to maximize the total amount of reward received by the agent over the long run.

Remark 1. *This definition is quite general: time can be continuous or discrete, with finite or infinite horizon; the state transitions can be stochastic or deterministic, the rewards can be stationary or not, and deterministic or sampled from a given distribution. In some cases (with an unknown model), the agent may start with partial or no knowledge about its environment.*

1.4 Definitions

Return. To maximize the long-term cumulative reward after the current time t, in the case of a finite time horizon that ends at time T, the *return* R_t is equal to:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T = \sum_{k=t+1}^{T} r_k$$

In the case of an infinite time horizon, it is customary instead to use a *discounted* return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

which will converge if we assume the rewards are bounded and $\gamma < 1$. Here $\gamma \in [0, 1]$ is a constant, called the *discount factor*. In what follows, in general we will use this discounted definition for the return.

Value Functions. In order to find an optimal policy, some algorithms are based on *value functions*, $V(s)$, that represent how beneficial it is for the agent to reach a given state s. Such a function provides, for each state, a numerical estimate of the potential future reward obtainable from this state, and thus depends on the actual policy π followed by the agent:

$$V^\pi(s) = \mathbb{E}_\pi \left[R_t \mid s_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right]$$

where $\mathbb{E}_\pi [.]$ denotes the expected value given that the agent follows policy π, and t is any time step.

Remark 2. *The existence and uniqueness of V^π are guaranteed if $\gamma < 1$ or if T is guaranteed to be finite from all states under the policy π [113].*

Action-Value Functions. Similarly, we define the value of taking action a in state s under a policy π as the *action-value function* Q:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi \left[R_t \mid s_t = s, a_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right] \end{aligned}$$

Optimal Policy. An *optimal policy* π^* is a policy that achieves the greatest expected reward over the long run. Formally, a policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. Thus:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} V^{\pi}(s) \quad \forall s \in \mathcal{S}$$

Remark 3. *There is always at least one policy that is better than or equal to all other policies. There may be more than one, but we denote all of them by* π^* *because they share the same value function and action-value function, noted:*

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in \mathcal{S}$$
$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad \forall s \in \mathcal{S}, \quad \forall a \in \mathcal{A}(s)$$

1.5 Markov Decision Processes (MDPs)

A *Markov Decision Process* is a particular instance of reinforcement learning where the set of states is finite, the sets of actions of each state are finite, and the environment satisfies the following Markov property:

$$Pr(s_{t+1} = s' | s_0, a_0, ...s_t, a_t) = Pr(s_{t+1} = s' | s_t, a_t)$$

In other words, the probability of reaching state s' from state s by action a is independent of the other actions or states in the past (before time t). Hence, we can represent a sequence of actions, states, rewards sampled from an MDP by a decision network (see Fig. 2).

Most reinforcement learning research is based on the formalism of MDPs. MDPs provide a simple framework in which to study basic algorithms and their properties. We will continue to use this formalism in Sect. 2. Then, we will emphasize its drawbacks in Sect. 3 and present potential improvements in Sect. 4.

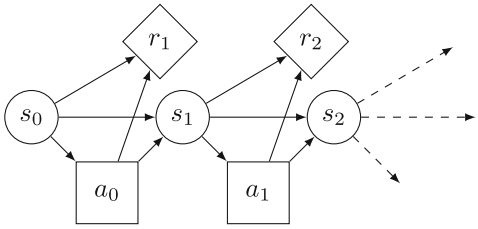

Fig. 2. Decision network representing an episode sampled from an MDP

1.6 A Visualization of Reinforcement Learning Algorithms

An overview of the algorithms that will be presented in this chapter can be found in Fig. 3. While this does not cover all reinforcement learning algorithms, we present it as a tool for the reader to get an overview of the reinforcement learning landscape. Each algorithm is color-coded according to whether it is *model based* or *model free*. Model based methods, such as those presented in Sects. 2.2 and 2.5, require a model of the environment while model free methods, such as those presented in Sects. 2.3 and 2.4, do not require a model of the environment. The functions (value function, action-value function, and/or policy function) that each algorithm uses are displayed beneath the algorithm. As shown in Sect. 5, these functions can take the form of deep neural networks.

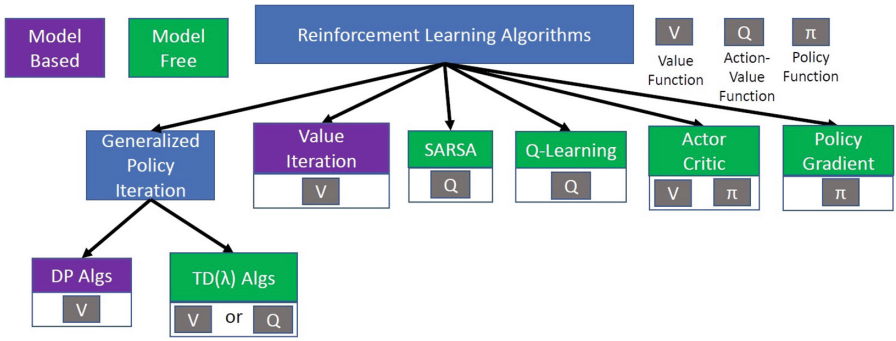

Fig. 3. An overview of the reinforcement learning algorithms that will be presented in this paper. The functions associated with each reinforcement learning algorithm can take the form of a deep neural network.

2 Main Algorithmic Approaches

Given a reinforcement learning problem, we now are going to present different approaches to computing the optimal policy. There are two main approaches: one based on searching in the space of value functions, and one based on searching in the space of policies. Value function space search methods attempt to compute the optimal value function V^* and deduce at the end the optimal policy π^* from V^*. These methods include linear programming, dynamic programming, Monte-Carlo methods, and temporal difference methods. Policy space search methods, on the other hand, maintain explicit representations of policies and update them over the time in order to compute the optimal policy π^*. Such methods typically include evolutionary and policy gradient algorithms. We provide a brief overview of these methods in the following sections.

2.1 Linear Programming

In order to cast the goal of finding the optimal value function as a linear programming problem [89], we treat the value function V as a cost function and then try to minimize the cost from each starting state s. In order to minimize a cost, we need to invert the sign of the rewards. We will note the cost function $g_\pi(s_t) = -r_{t+1}$. Thus here we want to minimize:

$$J^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k g_\pi(s_k) \middle| s_0 = s \right]$$

In order to perform this minimization, we define the optimal Bellman operator T:

$$(TJ)(s) = \min_\pi (g_\pi(s) + \gamma P_\pi(s) J)$$

where J is a vector of states, P_π is the transition matrix with the (s, s') entry representing the probability of reaching s' from s under policy π, and the minimization is carried out component-wise.

The solution that minimizes the cost should verify the Bellman equation:

$$J(s) = (TJ)(s)$$

It can be found by solving the linear programming optimization (using, for example, the simplex algorithm):

$$\begin{aligned} &\min_J \quad \mu^T J \\ &\quad s.t.\ TJ \geq J \end{aligned}$$

where μ is a vector of positive weights, known as the *state-relevance weights*.

From a theoretical perspective, linear programming provides the only known algorithm that can solve MDPs in polynomial time, although in general linear programming approaches to reinforcement learning problems do not fare well in practice. In particular, the main problem for linear programming approaches is that the time and space complexity can be extremely high.

2.2 Dynamic Programming

Dynamic programming algorithms are the simplest way to tackle a reinforcement learning problem, however, this method requires perfect knowledge of the model and is limited by its computational cost. The idea behind the dynamic programming formulation of reinforcement learning is to choose a policy π, estimate its value function V^π (Algorithm 1), deduce a new policy π' from V^π (Algorithm 2), and iterate this process until a satisfying policy is found (Algorithm 3). This process is known as *policy iteration*. Since each step strictly improves the policy, the algorithm is guaranteed to converge to the optimal policy. For computational convenience, one can decide to stop the policy evaluation step when the change in the value function is small between two iterations, as implemented below with the threshold θ:

Algorithm 1. Policy Evaluation

Data: π, the policy to be evaluated
Result: $V \approx V^{\pi}$, an approximation of the value function of π
repeat
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a(s,s')(R^a(s,s') + \gamma V(s'))$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$;

Remark 4. *At each step k, the value function V_{k+1} can be computed from the previous one V_k in two ways [113]:*

- *Full Backup: using two distinct arrays to store the two functions V_k and V_{k+1}.*
- *In Place: using only one array, and overwriting V_k when computing V_{k+1} for each state.*

The second approach is usually faster.

Algorithm 2. Policy Improvement

Data: π, the policy to be updated
  V, the value function
Result: π, the updated policy
for $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \underset{a}{\text{argmax}} \sum_{s'} P^a(s,s')(R^a(s,s') + \gamma V(s'))$

Algorithm 3. Policy Iteration

Result: π^*, the optimal policy
Initialization: π chosen arbitrarily
repeat
  $\pi_0 \leftarrow \pi$
  $V = \text{Policy\_evaluation}(\pi)$
  $\pi = \text{Policy\_improvement}(\pi, V)$
until $\pi_0 = \pi$;

One drawback of policy iteration is the policy evaluation step; which requires multiple iterations over every state. Another way to proceed is to combine policy evaluation and policy improvement in the same loop (Algorithm 4). This process is called value iteration. Value iteration is not always better than policy iteration, the efficiency depends on the nature of the problem and the parameters chosen. These differences are discussed in [85].

Algorithm 4. Value Iteration

Result: π^*, the optimal policy
Initialization: V chosen arbitrarily
repeat
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P^a(s, s')(R^a(s, s') + \gamma V(s'))$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$;
for $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s'} P^a(s, s')(R^a(s, s') + \gamma V(s'))$

2.3 Monte-Carlo Methods

The following algorithms correspond to online learning methods that do not require any knowledge of the environment. To estimate the value function V^π of a policy π, we must generate a sequence of actions and states with π, called an *episode*, compute the total reward at the end of this sequence, then update the estimate of V^π, V, for each state of the episode according to its contribution to the final reward, and repeat this process. One way to achieve this is to compute the average of the expected return from each state (Algorithm 5).

When one has a model of the environment, state values alone are sufficient to determine a policy. At any state s, the action taken is:

$$\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s'} P^a(s, s')(R^a(s, s') + \gamma V(s'))$$

However, without a model, we will not have access to the state transition probabilities and/or the expected reward; therefore, we will not be able to find action a that maximizes the aforementioned expression. Therefore, action-value

Algorithm 5. MC Policy Evaluation

Data: π, the policy to be evaluated
Result: $V \approx V^\pi$, an approximation of the value function of π
Initialization: V chosen arbitrarily
  $Returns(s) \leftarrow [], \forall s \in S$
repeat
  $episode = generate\_episode(\pi)$
  for $s \in episode$ **do**
    $R \leftarrow$ Return following first occurrence of s
    $Returns(s).append(R)$
    $V(s) \leftarrow \text{average}(Returns(s))$
until;

Algorithm 6. MC Exploring Starts

Result: π^*, the optimal policy
Initialization: Q chosen arbitrarily
π chosen arbitrarily
$Returns(s, a) \leftarrow []$, $\forall s \in S$, $\forall a \in \mathcal{A}(s)$
repeat
  episode = generate_episode_exploring_starts(π)
  for $s, a \in$ *episode* **do**
    $R \leftarrow$ Return following first occurrence of s, a
    $Returns(s, a).append(R)$
    $Q(s, a) \leftarrow$ average($Returns(s, a)$)
  for $s \in$ *episode* **do**
    $\pi(s) \leftarrow \underset{a}{\text{argmax}}\, Q(s, a)$
until;

functions are necessary to find the optimal policy. If we are following a deterministic policy, many state-action pairs may never be visited. We present two different methods for addressing this problem: *exploring starts* [113] and *stochastic policies*. Similar to value iteration, the methods we present for *exploring starts* and *stochastic policies* do not wait to complete policy evaluation before doing policy improvement. Instead, policy evaluation and policy improvement are done every episode.

Under the *exploring starts* assumption, each episode starts at a state-action pair and every state-action pair has a nonzero chance of being the starting pair. This algorithm is shown in Algorithm 6.

The exploring starts assumption may often be infeasible in practice. To explore as many state-action pairs as possible, one must consider policies that are stochastic. We distinguish between two different types of policies: The policy that is used to generate episodes (the behavior policy) and the policy that is being evaluated and improved (the estimation policy). The behavior policy must be stochastic in order to ensure new state-action pairs are explored. There are two main types of methods that utilize *stochastic policies*: *on-policy* methods and *off-policy* methods. For *on-policy* methods, the behavior policy and the estimation policy are the same; therefore, the policy that is being evaluated and improved must also be stochastic. Algorithm 7 shows an *on-policy* MC algorithm that utilizes an ϵ-*greedy* policy: with probability ϵ it chooses an action at random, otherwise, it chooses the greedy action.

On the other hand, *off-policy* methods can have a behavior policy that is separate from the estimation policy. The behavior policy should still be stochastic and must have a nonzero probability of selecting all actions that the estimation policy might select, however, the estimation policy can be greedy and always select the action a at state s that maximizes $Q(s, a)$. The downside of *off-policy* methods is that policy improvement is slower because it can only learn from states where the behavior policy and the estimation policy take the same

Algorithm 7. MC On-Policy Control

Result: π^*, the optimal policy
Initialization: Q chosen arbitrarily
π chosen arbitrarily
$Returns(s, a) \leftarrow [], \forall s \in S, \ \forall a \in \mathcal{A}(s)$
repeat
  episode = generate_episode(π)
  for $s, a \in$ *episode* **do**
    $R \leftarrow$ Return following first occurrence of s, a
    $Returns(s, a).append(R)$
    $Q(s, a) \leftarrow$ average($Returns(s, a)$)
  for $s \in$ *episode* **do**
    $a^* \leftarrow \underset{a}{\text{argmax}}\, Q(s, a)$
    for $a \in \mathcal{A}(s)$ **do**
      $\pi(s, a) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \epsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
until;

actions. Differences between *on-policy* and *off-policy* methods are discussed further in [113]. A well-known off-policy algorithm, Q-learning, will be presented in Sect. 2.4.

Remark 5. *The MC methods presented in this paper are first-visit MC methods. The first-visit method averages the return following the first visit to a state s in an episode, in the case of MC policy evaluation, or following the first occurrence of the state-action pair s, a, in the case of MC exploring starts and MC on-policy control. There are also every-visit methods that use the return from every occurrence of s or s, a. However, these methods are less straightforward because of the introduction of bias [106].*

2.4 Temporal Difference Methods

TD(0). Whereas the Monte-Carlo algorithms are constrained to wait for the end of an episode to update the value function, the TD(0) algorithm (Algorithm 8) is able to compute an update after every step:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]$$

When working with action-value functions, a well-known *off-policy* algorithm known as Q-learning (Algorithm 9) approximates Q^* regardless of the current policy.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

Algorithm 8. TD(0)

Data: π, the policy to be evaluated
Result: $V \approx V^{\pi}$, an approximation of the value function of π
Initialization: V chosen arbitrarily
repeat
  $s \leftarrow get\_initial\_state()$
  while s *not terminal* **do**
    $a \leftarrow get\_action(\pi, s)$
    $s', r \leftarrow get\_next\_state(s, a)$
    $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$
    $s \leftarrow s'$
until;

Algorithm 9. Q-Learning

Result: π^*, the optimal policy
Initialization: Q chosen arbitrarily
repeat
  $s \leftarrow get\_initial\_state()$
  while s *not terminal* **do**
    $a \leftarrow get\_action(Q, s)$
    $s', r \leftarrow get\_next\_state(s, a)$
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
until;

Remark 6. *An on-policy variant of the Q-Learning algorithm, called the SARSA algorithm [90], consists of choosing a' with respect to the current policy for selecting the next action, rather than the max of the value function for the next state.*

TD(λ) [forward view]. The TD(λ) algorithm, with λ chosen between 0 and 1, is a compromise between the full backup method of the Monte-Carlo algorithm and the step-by-step update of the TD(0) algorithm. It relies on backups of episodes that are used to update each state, while assigning a greater importance to the very next step after each state.

We first define a n-step target: $R_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} r_{t+k} + \gamma^n V(s_{t+n})$ Then, we can introduce the particular averaging of the TD(λ) algorithm on a state at time t in an episode ending at time T:

$$R_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

This can be expanded as:

$$\begin{aligned} R_t^\lambda = &(1-\lambda)(r_{t+1} + \gamma V(s_{t+1})) \\ &+(1-\lambda)\lambda(r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2})) \\ &+(1-\lambda)\lambda^2(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V(s_{t+2})) \\ &... \\ &+\lambda^{T-t-1}(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-1} r_T) \end{aligned}$$

Finally, the update method used is:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[R_t^\lambda - V(s_t)\right]$$

Remark 7. *One can notice that the sum of the weights* $(1-\lambda)\lambda^{n-1}$ *and* λ^{T-t-1} *is equal to* 1*. Moreover:*

- *If* $\lambda = 0$*, the algorithm corresponds to TD(0).*
- *If* $\lambda = 1$*, the algorithm corresponds to the MC algorithm.*

TD(λ)[backward view]. The previous description of TD(λ) illustrates the mechanism behind this method. However, it is not computationally tractable. Here, we describe an equivalent approach that leads to an efficient implementation.

We have to introduce for each state the *eligibility trace* $e_t(s)$ that represents how much the state will influence the update of a future encountered state in an episode:

$$e_t(s) = \begin{cases} 0 & \text{if } t = 0 \\ \gamma\lambda e_{t-1}(s) & \text{if } t > 0 \text{ and } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } t > 0 \text{ and } s = s_t \end{cases}$$

We can now define the update method to be applied at each step t to all states s_i:

$$V(s_i) \leftarrow V(s_i) + \alpha e_t(s_i)\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

yielding Algorithm 10.

Actor-Critic Methods. Actor-Critic methods separate the policy and the value function into two distinct structures [54]. The *actor*, or policy structure, is used to select actions; while the *critic*, or the estimated value function V, is used to criticize those actions in the form of a TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Algorithm 10. TD(λ)

Result: $V \approx V^{\pi}$, an approximation of the value function of π
Initialization: V chosen arbitrarily
$e(s) = 0, \quad \forall s \in \mathcal{S}$
repeat
  $s \leftarrow$ get_initial_state()
  while $s \notin$ *Terminal* **do**
    $a \leftarrow$ get_action(π, s)
    $s', r \leftarrow$ *get_next_state*(s, a)
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for $u \in \mathcal{S}$ **do**
      $V(u) \leftarrow V(u) + \alpha \delta e(u)$
      $e(u) \leftarrow \gamma \lambda e(u)$
    $s \leftarrow s'$
until;

A positive δ_t indicates that the policy's decision to take action a_t in state s_t should be strengthened, on the other hand, a negative δ_t indicates that the policy's decision should be weakened. In a simple case, if the policy for s_t and a_t is just a scalar $p(s_t, a_t)$ that is then normalized across all actions (i.e. using a softmax function), we can adjust the parameters of the policy using δ_t:

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta \delta_t (1 - \pi_t(s_t, a_t))$$

where β is a positive scaling factor.

If $\pi_t(s_t, a_t)$ is a more complicated parameterized function, such as a deep neural network, then δ_t is used for computing gradients.

2.5 Planning

The key difference between dynamic programming methods and temporal difference methods is the use of a model. Dynamic programming methods use a model of the world to update the value of each state based on state transition probabilities and expectations of rewards. However, temporal difference methods achieve this through directly interacting with the environment.

A model produces a prediction about the future state and reward given a state-action pair. There are two main types of models: *distribution models* and *sample models*. A distribution model, like the one used in dynamic programming methods, produces all the possible next states with their corresponding probabilities and expected rewards, whereas a sample model only produces a sample next state and reward. Distribution models are more powerful than sample models; however, sample models can be more efficient in practice [113].

The benefit of a model is that one can simulate interactions with the environment, which is usually less costly than interacting directly with the environment itself. The downside is that a perfect model does not always exist. A model may have to be approximated by hand or learned through real-world interaction with the environment. Any sub-optimal behavior in the model can lead to a sub-optimal policy. [112] presented an algorithm that combines reinforcement learning, model learning, and planning (Algorithm 11) [113]. This algorithm requires that the environment be deterministic. The resulting state and reward of each observed state-action pair is stored in the model. The agent can then use the model to improve the action-values associated with each previously seen state-action pair without having to interact with the environment.

Algorithm 11. Dyna-Q

Result: π^*, the optimal policy
Initialization: Q chosen arbitrarily
$Model(s, a)$ chosen arbitrarily $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$
N some positive integer
repeat
  $s \leftarrow$ current (nonterminal) state
  $a \leftarrow get\_action(Q, s)$
  $s', r \leftarrow get\_next\_state(s, a)$
  $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
  $Model(s, a) \leftarrow s', r$
  $n \leftarrow 0$
  repeat
    $s \leftarrow$ random previously seen state
    $a \leftarrow$ random action previously taken in s
    $s', r \leftarrow Model(s, a)$
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $n \leftarrow n + 1$
  until $n >= N$;
until;

A model can be used to improve a value function and policy or it can be used to pick better actions given the current value function and policy. Heuristic search does this by using the value function and policy as a "heuristic" to search the state-space in order to select better actions. Monte Carlo tree search (MCTS) [18,53] is a heuristic search algorithm which uses a model to run simulations from the current state. When searching the state-space, the probability of selecting an action a in state s is influenced by the policy as well as the number of times that state-action pair has been selected. In order to encourage exploration, the probability of selecting a state-action pair goes down each time that pair is selected. Backed up values come from either running the simulation until the end of the episode or from the value of the leaf nodes.

2.6 Evolutionary Algorithms

We now turn to algorithms that search the policy space, starting with evolutionary algorithms. These algorithms mimic the biological evolution of populations under natural selection (see [74] for more details). In reinforcement learning applications, populations of policies are evolved using a *fitness function*. At each generation, the most fit policies have a better chance of surviving and producing offspring policies in the next generation.

The most straightforward way to represent a policy in an evolutionary algorithm is to use a single chromosome per policy, with a single gene associated with each observed state. Each allele (the value of a gene) represents the action-value associated with the corresponding state. The algorithm (Algorithm 12) first generates a population of policies $P(0)$, then selects the best ones according to a given criteria (*selection*), then randomly perturbs these policies (for instance by randomly selecting a state and then randomly perturbing the distribution of the actions given that state) (*mutation*). The algorithm may also create new policies by merging two different selected policies (*crossover*). This process is repeated until the selected policies satisfy a given criteria.

The fitness of a policy in the population is defined as the expected accumulated rewards for an agent that uses that policy. During the selection step, we keep either the policies with the highest fitness, or use a probabilistic choice in order to avoid local optima, such as:

$$Pr(p_i) = \frac{\text{fitness}(p_i)}{\sum_{j=1}^{n} \text{fitness}(p_j)}$$

Algorithm 12. Evolutionary Algorithm

Result: $\pi \approx \pi^*$, an approximation of the optimal policy
Initialization: $t = 0$
population $P(0)$ chosen arbitrarily
repeat
  $t \leftarrow t + 1$
  select $P(t)$ from $P(t-1)$
  apply_mutation($P(t)$)
  apply_crossover($P(t)$)
until;

2.7 Policy Gradient Algorithms

While other approaches tend to struggle with large or continuous state spaces, policy gradient algorithms offer a good alternative for complex environments solvable by relatively simple policies. Starting with an arbitrary policy, the idea behind policy gradient is to modify the policy such that it obtains the largest

reward possible. For this purpose, a policy is represented by a parametric probability distribution $\pi_\theta(a|s) = P(a|s, \theta)$ such that in state s action a is selected according to the distribution $P(a|s, \theta)$. Hence, the objective here is to tune the parameter θ to increase the probability of choosing episodes associated with greater rewards. By computing the gradient of the average total return of a batch of episodes sampled from π_θ, we can use this value to update θ step-by-step. This approach is exploited in the REINFORCE algorithm [126].

3 Limitations and Open Problems

3.1 Complexity Considerations

So far, we have presented several ways of tackling the reinforcement learning problem in the framework of MDPs, but we have not described the theoretical tractability of this problem.

Recall that **P** is the class of all problems that can be solved in polynomial time, and **NC** the class of the problems that can be solved in polylogarithmic time on a parallel computer with a polynomial number of processors. As it seems very unlikely that $\mathbf{NC} = \mathbf{P}$, if a problem is proved to be **P**-complete, one can hardly expect to be able to find a parallel solution to this problem. In particular, it has been proved that the MDP problem is **P**-complete in the case of probabilistic transitions, and is in **NC** in the case of deterministic transitions, by [82]. Furthermore, in the case of high-dimensional MDPs, there exists a randomized algorithm [50] that is able to compute an arbitrary near-optimal policy in time independent of the number of states.

Remark 8. *Note that* $\boldsymbol{NC} \subseteq \boldsymbol{P}$*, simply because parallel computers can be simulated on a sequential machine.*

Other results for the POMDP framework (see Sect. 3.3) are presented in [64]. In particular:

- Computing an infinite (polynomial) horizon undiscounted optimal strategy for a deterministic POMDP is PSPACE-hard (NP-complete).
- Computing an infinite (polynomial) horizon undiscounted optimal strategy for a stochastic POMDP is EXPTIME-hard (PSPACE-complete).

3.2 Limitations of Markov Decision Processes (MDPs)

Despite its great convenience as a theoretical model, the MDP model suffers from major drawbacks when it comes to real-world implementations. Here we list the most important ones to highlight common pitfalls encountered in practical applications.

- **High-dimensional spaces.** For high-dimensional spaces, typical of real-world control tasks, using a simple reinforcement learning framework becomes computationally intractable: this phenomenon is known as the *curse of dimensionality*. We can limit this by reducing the dimensionality of the problem [120], or by replacing the lookup table by a *function approximator* [15]. However, some precautions may need to be taken to ensure convergence [11].
- **Continuous spaces.** A variety of real world problems lead to continuous state spaces or action spaces, yet it is not possible to store an arbitrary continuous function. To address this problem, one has to use *function approximators* [94] to obtain tractable models, value functions, or policies. Two common techniques are *tile coding* [98] and fuzzy representation of the space [62].
- **Convergence.** While we have good guarantees on the convergence of reinforcement learning methods with lookup tables and linear approximators, our knowledge of the conditions for convergence with non-linear approximators is still very limited [119]. This is unfortunate because non-linear approximators are the most convenient and have been very successful on problems like playing backgammon [117].
- **Speed.** One way to speed up the convergence of reinforcement learning algorithms is to modify the reward function during learning to provide guidance toward good policies. This technique, called *shaping*, has been successfully applied to the problem of bike riding, which would not have been tractable without this improvement [88].
- **Stability.** Highly dependent on the parameters, the stability of the process of computing an optimal policy has not been studied sufficiently. However, it is a key element in the success of a learning strategy. Stability and stability guarantees have been studied in the context of kernel-based reinforcement learning methods [81].
- **Exploration vs Exploitation.** To learn efficiently, an agent in general should navigate the tradeoff between exploration and exploitation. Common heuristics such as ϵ-greedy and Boltzmann (softmax) provide means for addressing this trade-off, yet suffer from major drawbacks in terms of convergence speed and implementation (the choice of the parameters is non-trivial). The R-max algorithm [13], relying on the *optimism under uncertainty* bias, and model-based Bayesian exploration [22] offer convenient alternatives for the *exploration-exploitation dilemma*.
- **Initialization.** The choice of the initial policy, or the initial value function, may influence not only whether the algorithm converges, but also the speed of convergence. In some cases, for example, choosing a random initialization leads to drastically long computational times. One way to tackle this issue is to learn first using a simpler but similar task, and then use this knowledge to influence the learning process of the main task. This is the core principle of *transfer learning* which can lead to significant improvements, as shown in [116].

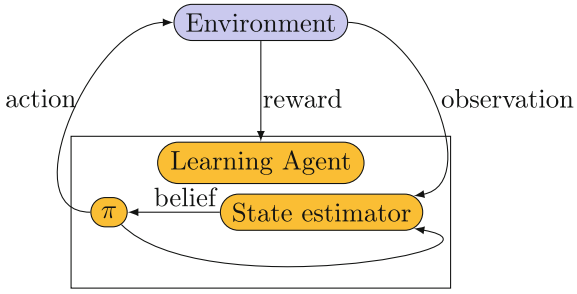

Fig. 4. The POMDP model

3.3 The POMDP Model

The partially observable Markov decision process (POMDP) [130] is a generalization of the MDP model in which the learning agent does not know precisely the current state in which it is operating. Instead, its knowledge relies on *observations* derived from its environment. Formally, a POMDP is an MDP with a finite set of possible observations $\mathcal{Z}$ and an observation model based on the probability $\nu(z|s)$ of observing z when the environment is in state s.

It has been shown in [105], that directly applying the MDP methods to this problem can have arbitrarily poor performance. To address this problem, one has to introduce an internal state distribution for the agent, the *belief state* $b_t(s)$, that represents the probability of being in state s at time t (see Fig. 4). One can then theoretically find an optimal solution to a POMDP problem [16] by defining an equivalent MDP problem, as shown below, and use existing MDP algorithms to solve it.

Assuming that the initial belief state b_0 is known, one can iteratively compute the belief state at any time $t+1$. We denote this operation by $F(b_t, a_t, z_t) = b_{t+1}$ with:

$$b_{t+1}(s') = \frac{\nu(z_t|s') \sum_{s \in \mathcal{S}} b_t(s) P_{a_t}(s, s')}{\sum_{s' \in \mathcal{S}} \nu(z|s') \sum_{s \in \mathcal{S}} b_t(s) P_{a_t}(s, s')}$$

The rewards are then given by:

$$\bar{r}(b) = \sum_{s \in \mathcal{S}} b(s) r(s)$$

In order to compute the transition function, let us first introduce the probability of observing z by applying action a in belief state b:

$$Pr(z|a, b) = \sum_{s' \in \mathcal{S}} \nu(z|s') \sum_{s \in \mathcal{S}} b(s) P_a(s, s')$$

Hence, we can define a transition probability function for the POMDP by:

$$\bar{P}_a(b, b') = \sum_{\substack{z \in \mathcal{Z} \\ F(b_t, a_t, z_t) = b_{t+1}}} Pr(z_t|a_t, b_t)$$

If $\mathcal{B}$ represents the set of belief states, the value function can then be computed as:

$$\bar{V}_{t+1}(b') = \max_a \left[\bar{r}(b') + \gamma \sum_{b \in \mathcal{B}} \bar{P}_a(b, b') \bar{V}_t(b) \right]$$

Remark 9. *This approach is obviously quite limited because of the potentially infinite size of $\mathcal{B}$. Several algorithms have been proposed to improve this, such as region-based pruning [31] and point-based algorithms [108], but they are also unable to deal with very large state spaces. VDCBPI [86] is one of the few efficient heuristics that seems to be able to find reasonable approximate solutions.*

3.4 Multi-agent Paradigm

There are several reasons for studying the case of multiple agents interacting with each other and seeking to maximize their rewards in a reinforcement learning fashion [14]. Many problems in areas as diverse as robotics, control, game theory, and population modeling lend themselves to such a modeling approach. Furthermore, the ability to parallelize learning across multiple agents is also attractive for several reasons, including speed and robustness. In particular, one may expect that if a particular agent fails, the other agents may be able to adapt without leading to a system-wide failure. Lastly, one may be able to improve or speed up learning of similar tasks by sharing experiences between individual learners (*transfer learning*).

However, as can be expected, the multi-agent model comes with significant challenges. By definition the multi-agent model has more variables and thus the curse of dimensionality is heightened. Furthermore, the environment model is more complex and suffers from non-stationarity during learning because of the constantly evolving behavior of each agent, and the problem of coordination between agents in order to achieve the desired results.

The starting model for the multi-agent paradigm corresponds to a *stochastic game*. For a system with n agents, it is composed of a set of states X, the sets of actions U_i for each agent $i = 1, ..., n$ (we let $\mathbf{U} = U_1 \times ... \times U_n$), the state transition function $f : X \times \mathbf{U} \times X \to [0, 1]$ and the reward function $\rho_i : X \times \mathbf{U} \times X \to \mathbb{R}$.

There is a large collection of literature with different methods suitable for different multi-agent settings. The two major characteristics of such algorithms are their stability, which is related to their ability to converge to a stationary policy, and their adaptation, which measures how well the agents react to a change in the policy. Usually, it is difficult to guarantee both, and one must favor one over the other. The relationships between the agents can be classified in several classes, including:

- Fully cooperative: all the agents share a common set of objectives that have to be maximized. The *optimal adaptive learning* algorithm [122] has been proven to converge to an optimal Nash equilibrium (a configuration where no agent can improve its expected payoff by deviating to a different strategy) with probability 1. Good experimental results have also been obtained with the *coordinated reinforcement learning* approach [36].

- Fully competitive: the success of each agent directly depends on the failure of the other agents. For such settings, the minimax-Q [63] algorithm has been proposed, combining the minimax strategy (acting optimally while considering that the adversary will also act optimally) with the Q-learning method.
- Mixed: each agent has its own goal. As the objectives of this scenario are not well defined, there exist a significant number of approaches designed to tackle various formulations of this setting. An attempt to organize and clarify this case has been proposed in [87], for instance, along with a comparison of the most popular methods.

4 Other Directions of Research

4.1 Inverse Reinforcement Learning

Inverse reinforcement learning is the task of determining the reward function given an observed behavior. This observed behavior can be an optimal policy or a teacher's demonstration. Thus, the objective here is to estimate the reward attribution such that when reinforcement learning is applied with that reward function, one obtains the original behavior (in the case of behaviors associated with optimal policies), or even a better one (in the case of demonstrations).

This is particularly relevant in a situation where an expert has the ability to execute a given task but is unable, due to the complexity of the task and the domain, to precisely define the reward attribution that would lead to an optimal policy. One of the most significant success stories of inverse reinforcement learning is the apprenticeship of self driving cars [1].

To solve this problem in the case of MDPs, [78] identifies inequalities such that any reward function satisfying them must lead to an optimal policy. In order to avoid trivial answers, such as the all-zero reward function, these authors propose to use linear programming to identify the reward function that would maximize the difference between the value of an optimal action and the value of the next-best action in the same state. It is also possible to add regularization on the reward function to make it simpler (typically with non-zero reward on few actions). Systematic applications of inverse reinforcement learning in the case of POMDPs have not yet been developed.

4.2 Hierarchical Reinforcement Learning

In order to improve the time of convergence of reinforcement learning algorithms, different approaches for reducing the dimensionality of the problem have been proposed. In some cases, these approaches extend the MDP model to semi-Markov Decision Process (SMDP), by relaxing the Markov property, i.e. policies may base their choices on more than just the current state.

The *option* method [114] makes use of local policies that focus on simpler tasks. Hence, along with actions, a policy π can choose an option O. When the option O is chosen, a special policy μ associated with O is followed until a

stochastic stop condition over the states and depending on O is reached. After the stop condition is reached, the policy π is resumed. The reward associated with O is the sum of the rewards of the actions performed under μ discounted by γ^τ were τ is the number of steps needed to terminate the option O. These option policies can be defined by an expert, or learned. There has been some work to try to automate this process of creating relevant options, or deleting useless ones [66].

State abstraction [4], used in the MAXQ algorithm [24] and in *hierarchical abstract machines* [83], is a mapping of the problem representation to a new representation that preserves some of its properties, in particular those needed for learning an optimal policy.

4.3 Approximate Linear Programming

As noted before, the linear programming approach to reinforcement learning typically suffers from the curse of dimensionality: the large number of states leads to an intractable number of variables for applying exact linear programming. A common way to overcome this issue is to approximate the cost-to-go function [30] by carefully designing some basis functions $\phi_1, ..., \phi_K$ that map the state space to rewards, and then constructing a linearly parameterized cost-to-go function:

$$\tilde{J}(\cdot, r) = \sum_{k=1}^{K} r_k \phi_k$$

where r is a parameter vector to be approximated by linear programming. In this way, the number of variables of the problem is drastically reduced, from the original number of states to K. The work in [45] proposes automated methods for generating a suitable basis functions ϕ for a given problem.

Using a *dynamic Bayesian network* to represent the transition model leads to the concept of *factored MDP* that can lead to reduced computational times on problems with a large number of states [35].

4.4 Relational Reinforcement Learning

Relational reinforcement learning [28] combines reinforcement learning with a relational representation of the state space, for instance by using inductive logic programming [75]. The goal is to propose a formalism that is able to perform well on problems requiring a large number of states, but can be represented compactly using a relational representation. In particular, experiments highlight the ability of this approach to take advantage of learning on simple tasks to accelerate the learning on more complex ones. This representation allows the learning of more "abstract" concepts, which leads to a reduced number of states that can significantly benefit generalization.

4.5 Quantum Reinforcement Learning

By taking advantage of the properties of quantum superposition, there is a possibility for considering novel quantum algorithms for reinforcement learning. The study in [25] presents potentially promising results, through simulated experiments, in regards to the speed of convergence and the trade-off between exploration and exploitation. Much work remains to be done in relation to modeling the environment, implementing function approximations, and deriving theoretical guarantees for quantum reinforcement learning (Fig. 5).

5 Deep Reinforcement Learning

Neural networks and deep learning approaches have well known universal approximation properties [21,43]. In recent years, and although they are far from new [96], neural networks and deep learning approaches have been used to successfully tackle a variety of problems in engineering, ranging from computer vision [5,20,38,55,109,115] to speech recognition [34], to natural language processing [32,107,110]. Likewise, deep learning is playing an essential role in the natural sciences, in areas ranging from high energy physics [7,92], to chemistry [48,49,65], and to biology [2,6,23,29,131]. Most of these applications use supervised, or semi-supervised learning, with stochastic gradient descent as the main learning algorithm and have benefited from significant increases in the amounts of available training data and computing power, including GPUs, as well as the development of good neural network software libraries. [71] also showed that, in certain cases, it is more efficient to train deep reinforcement learning algorithms using many CPUs instead of just one GPU.

It is therefore natural to try to combine deep learning methods with reinforcement learning methods, possibly in combination with frameworks for massively distributed reinforcement learning, such as Gorila [76]. This has been done, for instance, for the game of Go. The early work in [127,128] used deep learning methods, in the form of recursive grid neural networks, to evaluate the board or decide the next move. One characteristic of this approach is the ability to transfer learning between different board sizes (e.g. learn from games played on 9×9 or 11×11 boards and transfer the knowledge to larger boards). More recently, reinforcement learning combined with massive convolutional neural networks has been used to achieve the AI milestone of building an automated Go player [100] that can outperform human experts. Thus, deep reinforcement learning is a very active current area of research.

5.1 Value-Based Deep Reinforcement Learning

For *value-based deep reinforcement learning*, the value function is approximated by a deep neural network. [72] used Deep Q-networks that combine Q-learning with such a neural representation in order to teach an agent to play Atari video games, without any game-specific feature engineering. In this case, the *state* is represented by the stack of four previous frames, with the deep network consisting of multiple convolutional and fully-connected layers, and the action consisting

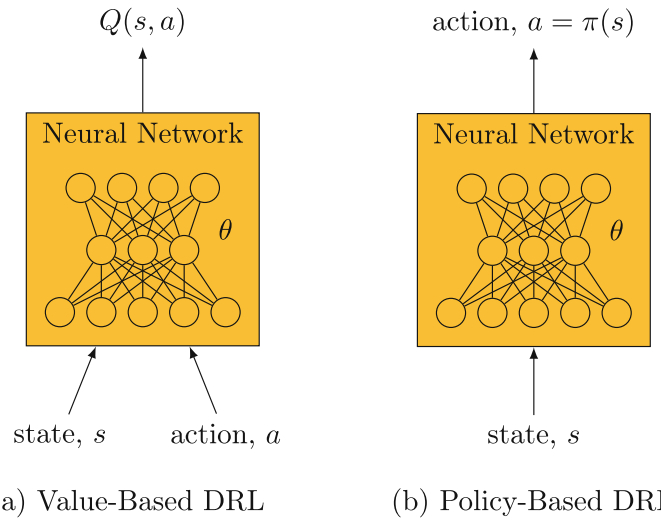

Fig. 5. Two types of deep reinforcement learning

of the 18 joystick positions. Since directly using neural networks as the function approximator leads to instability or divergence, the authors used additional heuristics such as replaying histories to reduce correlations, or using updates that change the parameters only periodically. Agents using this approach learned to play the majority of the games at a level equal or higher than the level of professional human players for the same game. There has been subsequent work to improve this approach, such as addressing stability [8], and applying Double Q-Learning [37] to avoid overestimation of the action-value functions in Deep Q-Networks [121]. Other extensions include multi-task learning [91,95], and rapid learning [10], among others.

5.2 Policy-Based Deep Reinforcement Learning

The second class of approaches, *policy-based deep reinforcement learning*, approximates the policy with deep neural networks. Policy-based approaches, by avoiding the search over the possible actions, converge and train much faster for many problems, especially with high-dimensional or continuous action spaces. The deterministic policy gradient, proposed by [102] and subsequently extended to deep representations by [61], were shown to be more efficient than their stochastic variants, thus extending deep reinforcement learning to continuous action spaces. [71] introduced the asynchronous advantage actor-critic (A3C) algorithm, that lets agents efficiently learn tasks with continuous action spaces, and works both on 2D and 3D games, with both feed-forward and recurrent neural approximators. As an application to robotic grasping, [60] uses a policy-gradient approach with a single deep convolutional network that combines the

visual input and the gripper motor control to predict the grasp success probability. For multi-agent reinforcement learning, deep reinforcement learning has been used to learn agents by combining the fictitious self-play (FSP) approach [40] with neural representations [41], and applied to games such as poker.

Of course, both value- and policy-based deep reinforcement learning can be combined together with search algorithms. This is precisely the approach used in [100] for the game of Go.

5.3 Planning with Deep Reinforcement Learning

In Algorithm 11, a look up table served as the model of the environment. However, it is intractable to represent high-dimensional environments, such as images, with a simple lookup table. To address this issue, deep neural networks have been trained to predict the next state and the reward given a state-action pair and thus, perform the task of the model. When the environment takes the form of an image, deep neural networks have been shown to be able to produce realistic images that the agent can use to plan [17,57,79,84,124,125]. However, the predicted images are sometimes noisy and are sometimes missing key elements of the state. An alternative approach is to use a deep neural network to encode the current state into an abstract state and then, given an action, learn to predict the next abstract state along with its value and reward [80,99].

In addition to improving action selection, heuristic search algorithms have been combined with value and policy networks to improve the value and policy networks themselves. When applying deep reinforcement learning to Go, [100] mainly used the MCTS algorithm for action selection while the value and policy networks relied heavily on gameplay from human experts. However, [103] used MCTS to train a value and policy network from scratch by using the heuristic search algorithm for self-play, which resulted in an agent that outperformed all previous Go agents. This approach was also used when learning to play chess and shogi [101].

Acknowledgment. This research was in part supported by National Science Foundation grant IIS-1550705 and a Google Faculty Research Award to PB.

References

1. Abbeel, P., Ng, A.Y.: Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the Twenty-First International Conference on Machine Learning, p. 1. ACM (2004)
2. Agostinelli, F., Ceglia, N., Shahbaba, B., Sassone-Corsi, P., Baldi, P.: What time is it? deep learning approaches for circadian rhythms. Bioinformatics **32**(12), i8–i17 (2016)
3. Anderson, C.W.: Learning to control an inverted pendulum using neural networks. Control Syst. Mag. IEEE **9**(3), 31–37 (1989)
4. Andre, D., Russell, S.J.: State abstraction for programmable reinforcement learning agents. In: AAAI/IAAI, pp. 119–125 (2002)

5. Baldi, P., Chauvin, Y.: Neural networks for fingerprint recognition. Neural Comput. **5**(3), 402–418 (1993)
6. Baldi, P., Pollastri, G.: The principled design of large-scale recursive neural network architectures-DAG-RNNs and the protein structure prediction problem. J. Mach. Learn. Res. **4**, 575–602 (2003)
7. Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nat. Commun. **5**, 4308 (2014)
8. Bellemare, M.G., Ostrovski, G., Guez, A., Thomas, P.S., Munos, R.: Increasing the action gap: new operators for reinforcement learning. In: AAAI, pp. 1476–1483 (2016)
9. Bellman, R.: The theory of dynamic programming. Technical report, DTIC Document (1954)
10. Blundell, C., et al.: Model-free episodic control. arXiv preprint arXiv:1606.04460 (2016)
11. Boyan, J., Moore, A.W.: Generalization in reinforcement learning: safely approximating the value function. In: Advances in Neural Information Processing Systems, pp. 369–376 (1995)
12. Boyan, J.A., Littman, M.L., et al.: Packet routing in dynamically changing networks: a reinforcement learning approach. In: Advances in Neural Information Processing Systems, pp. 671–671 (1994)
13. Brafman, R.I., Tennenholtz, M.: R-max-a general polynomial time algorithm for near-optimal reinforcement learning. J. Mach. Learn. Res. **3**, 213–231 (2003)
14. Busoniu, L., Babuska, R., De Schutter, B.: A comprehensive survey of multiagent reinforcement learning. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. **38**(2), 156–172 (2008)
15. Busoniu, L., Babuska, R., De Schutter, B., Ernst, D.: Reinforcement Learning and Dynamic Programming Using Function Approximators, vol. 39. CRC Press, Boca Raton (2010)
16. Cassandra, A.R., Kaelbling, L.P., Littman, M.L.: Acting optimally in partially observable stochastic domains. In: AAAI, vol. 94, p. 1023–1028 (1994)
17. Chiappa, S., Racaniere, S., Wierstra, D., Mohamed, S.: Recurrent environment simulators. arXiv preprint arXiv:1704.02254 (2017)
18. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75538-8_7
19. Crites, R., Barto, A.: Improving elevator performance using reinforcement learning. In: Advances in Neural Information Processing Systems, vol. 8. Citeseer (1996)
20. Cun, Y.L., et al.: Handwritten digit recognition with a back-propagation network. In: Touretzky, D. (ed.) Advances in Neural Information Processing Systems, pp. 396–404. Morgan Kaufmann, San Mateo (1990)
21. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Math. Control Signals Syst. (MCSS) **2**(4), 303–314 (1989)
22. Dearden, R., Friedman, N., Andre, D.: Model based Bayesian exploration. In: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, pp. 150–159. Morgan Kaufmann Publishers Inc. (1999)
23. Di Lena, P., Nagata, K., Baldi, P.: Deep architectures for protein contact map prediction. Bioinformatics **28**, 2449–2457 (2012). https://doi.org/10.1093/bioinformatics/bts475. First published online: July 30, 2012

24. Dietterich, T.G.: An overview of MAXQ hierarchical reinforcement learning. In: Choueiry, B.Y., Walsh, T. (eds.) SARA 2000. LNCS (LNAI), vol. 1864, pp. 26–44. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44914-0_2
25. Dong, D., Chen, C., Li, H., Tarn, T.J.: Quantum reinforcement learning. IEEE Trans. Syst. Man Cybern. Part B Cybern. **38**(5), 1207–1220 (2008)
26. Dorigo, M., Gambardella, L.: Ant-Q: a reinforcement learning approach to the traveling salesman problem. In: Proceedings of ML-95, Twelfth International Conference on Machine Learning, pp. 252–260 (2014)
27. Drake, A.W.: Observation of a Markov process through a noisy channel. Ph.D. thesis, Massachusetts Institute of Technology (1962)
28. Džeroski, S., De Raedt, L., Driessens, K.: Relational reinforcement learning. Mach. Learn. **43**(1–2), 7–52 (2001)
29. Esteva, A., et al.: Dermatologist-level classification of skin cancer with deep neural networks. Nature **542**(7639), 115–118 (2017)
30. de Farias, D.P., Van Roy, B.: The linear programming approach to approximate dynamic programming. Oper. Res. **51**(6), 850–865 (2003)
31. Feng, Z., Zilberstein, S.: Region-based incremental pruning for POMDPs. In: Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, pp. 146–153. AUAI Press (2004)
32. Goldberg, Y.: A primer on neural network models for natural language processing. J. Artif. Intell. Res. **57**, 345–420 (2016)
33. Gosavi, A.: Reinforcement learning: a tutorial survey and recent advances. INFORMS J. Comput. **21**(2), 178–192 (2009)
34. Graves, A., Mohamed, A., Hinton, G.: Speech recognition with deep recurrent neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6645–6649. IEEE (2013)
35. Guestrin, C., Koller, D., Parr, R., Venkataraman, S.: Efficient solution algorithms for factored MDPs. J. Artif. Intell. Res. **19**, 399–468 (2003)
36. Guestrin, C., Lagoudakis, M., Parr, R.: Coordinated reinforcement learning. In: ICML, vol. 2, pp. 227–234 (2002)
37. Hasselt, H.V.: Double q-learning. In: Advances in Neural Information Processing Systems, pp. 2613–2621 (2010)
38. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385 (2015)
39. Hebb, D.O.: The Organization of Behavior: A Neuropsychological Approach. Wiley, New York (1949)
40. Heinrich, J., Lanctot, M., Silver, D.: Fictitious self-play in extensive-form games. In: International Conference on Machine Learning (ICML), pp. 805–813 (2015)
41. Heinrich, J., Silver, D.: Deep reinforcement learning from self-play in imperfect-information games. arXiv preprint arXiv:1603.01121 (2016)
42. Holland, J.H.: Genetic algorithms and the optimal allocation of trials. SIAM J. Comput. **2**(2), 88–105 (1973)
43. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**(5), 359–366 (1989)
44. Howard, R.A.: Dynamic programming and Markov processes (1960)
45. Hutter, M.: Feature reinforcement learning: Part I. Unstructured MDPs. J. Artif. Gen. Intell. **1**(1), 3–24 (2009)
46. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
47. Kandel, E.R., Schwartz, J.H., Jessell, T.M.: Principles of Neural Science, vol. 4. McGraw-hill, New York (2000)

48. Kayala, M., Azencott, C., Chen, J., Baldi, P.: Learning to predict chemical reactions. J. Chem. Inf. Model. **51**(9), 2209–2222 (2011)
49. Kayala, M., Baldi, P.: Reactionpredictor: prediction of complex chemical reactions at the mechanistic level using machine learning. J. Chem. Inf. Model. **52**(10), 2526–2540 (2012)
50. Kearns, M., Mansour, Y., Ng, A.Y.: A sparse sampling algorithm for near-optimal planning in large Markov decision processes. Mach. Learn. **49**(2–3), 193–208 (2002)
51. Keerthi, S.S., Ravindran, B.: A tutorial survey of reinforcement learning. Sadhana **19**(6), 851–889 (1994)
52. Kober, J., Bagnell, J.A., Peters, J.: Reinforcement learning in robotics: a survey. Int. J. Robot. Res. **32**, 1238–1274 (2013). p. 0278364913495721
53. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_29
54. Konda, V.R., Tsitsiklis, J.N.: Actor-critic algorithms. In: NIPS. **13**, 1008–1014 (1999)
55. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
56. Lai, M.: Giraffe: Using deep reinforcement learning to play chess. arXiv preprint arXiv:1509.01549 (2015)
57. Leibfried, F., Kushman, N., Hofmann, K.: A deep learning approach for joint video frame and reward prediction in atari games. arXiv preprint arXiv:1611.07078 (2016)
58. Levin, E., Pieraccini, R., Eckert, W.: A stochastic model of human-machine interaction for learning dialog strategies. IEEE Trans. Speech Audio Process. **8**(1), 11–23 (2000)
59. Levine, S., Finn, C., Darrell, T., Abbeel, P.: End-to-end training of deep visuomotor policies. J. Mach. Learn. Res. **17**(39), 1–40 (2016)
60. Levine, S., Pastor, P., Krizhevsky, A., Quillen, D.: Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. In: International Symposium on Experimental Robotics (2016)
61. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning (2016)
62. Lin, C.T., Lee, C.G.: Reinforcement structure/parameter learning for neural-network-based fuzzy logic control systems. IEEE Trans. Fuzzy Syst. **2**(1), 46–63 (1994)
63. Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning. In: Proceedings of the Eleventh International Conference on Machine Learning, vol. 157, pp. 157–163 (1994)
64. Littman, M.L.: Algorithms for sequential decision making. Ph.D. thesis, Brown University (1996)
65. Lusci, A., Pollastri, G., Baldi, P.: Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules. J. Chem. Inf. Model. **53**(7), 1563–1575 (2013)
66. McGovern, A., Barto, A.G.: Automatic discovery of subgoals in reinforcement learning using diverse density. Computer Science Department Faculty Publication Series, p. 8 (2001)
67. Michie, D.: Trial and error. In: Science Survey, Part 2, pp. 129–145 (1961)
68. Michie, D.: Experiments on the mechanization of game-learning part I. Characterization of the model and its parameters. Comput. J. **6**(3), 232–236 (1963)

69. Michie, D., Chambers, R.A.: Boxes: an experiment in adaptive control. Mach. Intell. **2**(2), 137–152 (1968)
70. Minsky, M.: Steps toward artificial intelligence. Proc. IRE **49**(1), 8–30 (1961)
71. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning. In: International Conference on Machine Learning (ICML) (2016)
72. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
73. Moody, J., Saffell, M.: Reinforcement learning for trading. In: Advances in Neural Information Processing Systems, pp. 917–923 (1999)
74. Moriarty, D.E., Schultz, A.C., Grefenstette, J.J.: Evolutionary algorithms for reinforcement learning. J. Artif. Intell. Res. (JAIR) **11**, 241–276 (1999)
75. Muggleton, S., De Raedt, L.: Inductive logic programming: theory and methods. J. Logic Program. **19**, 629–679 (1994)
76. Nair, A., et al.: Massively parallel methods for deep reinforcement learning. arXiv preprint arXiv:1507.04296 (2015)
77. Ng, A.Y., et al.: Autonomous inverted helicopter flight via reinforcement learning. In: Ang, M.H., Khatib, O. (eds.) Experimental Robotics IX. STAR, vol. 21, pp. 363–372. Springer, Heidelberg (2006). https://doi.org/10.1007/11552246_35
78. Ng, A.Y., Russell, S.J., et al.: Algorithms for inverse reinforcement learning. In: ICML, pp. 663–670 (2000)
79. Oh, J., Guo, X., Lee, H., Lewis, R.L., Singh, S.: Action-conditional video prediction using deep networks in atari games. In: Advances in Neural Information Processing Systems, pp. 2863–2871 (2015)
80. Oh, J., Singh, S., Lee, H.: Value prediction network. In: Advances in Neural Information Processing Systems, pp. 6120–6130 (2017)
81. Ormoneit, D., Sen, Ś.: Kernel-based reinforcement learning. Mach. Learn. **49**(2–3), 161–178 (2002)
82. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of Markov decision processes. Math. Oper. Res. **12**(3), 441–450 (1987)
83. Parr, R., Russell, S.: Reinforcement learning with hierarchies of machines. In: Advances in Neural Information Processing Systems, pp. 1043–1049 (1998)
84. Pascanu, R., et al.: Learning model-based planning from scratch. arXiv preprint arXiv:1707.06170 (2017)
85. Pashenkova, E., Rish, I., Dechter, R.: Value iteration and policy iteration algorithms for Markov decision problem. In: AAAI 1996, Workshop on Structural Issues in Planning and Temporal Reasoning. Citeseer (1996)
86. Poupart, P., Boutilier, C.: VDCBPI: an approximate scalable algorithm for large POMDPs. In: Advances in Neural Information Processing Systems, pp. 1081–1088 (2004)
87. Powers, R., Shoham, Y.: New criteria and a new algorithm for learning in multi-agent systems. In: Advances in Neural Information Processing Systems, pp. 1089–1096 (2004)
88. Randløv, J., Alstrøm, P.: Learning to drive a bicycle using reinforcement learning and shaping. In: ICML, vol. 98, pp. 463–471. Citeseer (1998)
89. Ross, S.M.: Introduction to Stochastic Dynamic Programming. Academic press, Norwell (2014))
90. Rummery, G.A., Niranjan, M.: On-line Q-learning using connectionist systems. University of Cambridge, Department of Engineering (1994)
91. Rusu, A.A., et al.: Policy distillation. In: International Conference on Learning Representations (ICLR) (2016)

92. Sadowski, P., Collado, J., Whiteson, D., Baldi, P.: Deep learning, dark knowledge, and dark matter. In: Journal of Machine Learning Research, Workshop and Conference Proceedings, vol. 42, pp. 81–97 (2015)
93. Samuel, A.L.: Some studies in machine learning using the game of checkers. II. Recent progress. IBM J. Res. Dev. **11**(6), 601–617 (1967)
94. Santamaría, J.C., Sutton, R.S., Ram, A.: Experiments with reinforcement learning in problems with continuous state and action spaces. Adapt. Behav. **6**(2), 163–217 (1997)
95. Schaul, T., Horgan, D., Gregor, K., Silver, D.: Universal value function approximators. In: International Conference on Machine Learning (ICML), pp. 1312–1320 (2015)
96. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
97. Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P.: High-dimensional continuous control using generalized advantage estimation. In: Proceedings of the International Conference on Learning Representations (ICLR) (2016)
98. Sherstov, A.A., Stone, P.: On continuous-action Q-learning via tile coding function approximation. Under Review (2004)
99. Silver, D., et al.: The predictron: end-to-end learning and planning. arXiv preprint arXiv:1612.08810 (2016)
100. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
101. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
102. Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms. In: International Conference on Machine Learning (ICML) (2014)
103. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)
104. Singh, S., Bertsekas, D.: Reinforcement learning for dynamic channel allocation in cellular telephone systems. In: Advances in Neural Information Processing Systems, pp. 974–980 (1997)
105. Singh, S.P., Jaakkola, T.S., Jordan, M.I.: Learning without state-estimation in partially observable Markovian decision processes. In: ICML, pp. 284–292 (1994)
106. Singh, S.P., Sutton, R.S.: Reinforcement learning with replacing eligibility traces. Mach. Learn. **22**(1–3), 123–158 (1996)
107. Socher, R., et al.: Recursive deep models for semantic compositionality over a sentiment treebank. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), vol. 1631, p. 1642. Citeseer (2013)
108. Spaan, M.T., Spaan, M.T.: A point-based POMDP algorithm for robot planning. In: 2004 IEEE International Conference on Robotics and Automation, Proceedings, ICRA 2004, vol. 3, pp. 2399–2404. IEEE (2004)
109. Srivastava, R.K., Greff, K., Schmidhuber, J.: Training very deep networks. In: Advances in Neural Information Processing Systems, pp. 2368–2376 (2015)
110. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
111. Sutton, R.S.: Learning to predict by the methods of temporal differences. Mach. Learn. **3**(1), 9–44 (1988)

112. Sutton, R.S.: Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: Machine Learning Proceedings 1990, pp. 216–224. Elsevier (1990)
113. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
114. Sutton, R.S., Precup, D., Singh, S.: Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. Artif. Intell. **112**(1), 181–211 (1999)
115. Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
116. Taylor, M.E., Stone, P.: Cross-domain transfer for reinforcement learning. In: Proceedings of the 24th International Conference on Machine Learning, pp. 879–886. ACM (2007)
117. Tesauro, G.: Temporal difference learning and TD-Gammon. Commun. ACM **38**(3), 58–68 (1995)
118. Thorndike, E.L.: Animal Intelligence: Experimental Studies. Transaction Publishers, New York (1965)
119. Tsitsiklis, J.N., Van Roy, B.: An analysis of temporal-difference learning with function approximation. IEEE Trans. Autom. Control **42**(5), 674–690 (1997)
120. Van Der Maaten, L., Postma, E., Van den Herik, J.: Dimensionality reduction: a comparative. J. Mach. Learn. Res. **10**, 66–71 (2009)
121. Van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. In: AAAI, pp. 2094–2100 (2016)
122. Wang, X., Sandholm, T.: Reinforcement learning to play an optimal Nash equilibrium in team Markov games. In: Advances in Neural Information Processing Systems, pp. 1571–1578 (2002)
123. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992)
124. Watter, M., Springenberg, J., Boedecker, J., Riedmiller, M.: Embed to control: a locally linear latent dynamics model for control from raw images. In: Advances in Neural Information Processing Systems, pp. 2746–2754 (2015)
125. Weber, T., et al.: Imagination-augmented agents for deep reinforcement learning. arXiv preprint arXiv:1707.06203 (2017)
126. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn. **8**(3–4), 229–256 (1992)
127. Wu, L., Baldi, P.: A scalable machine learning approach to go. In: Weiss, Y., Scholkopf, B., Editors, J.P. (eds.) NIPS 2006. MIT Press, Cambridge (2007)
128. Wu, L., Baldi, P.: Learning to play go using recursive neural networks. Neural Netw. **21**(9), 1392–1400 (2008)
129. Zhang, W., Dietterich, T.G.: High-performance job-shop scheduling with a time-delay td network. In: Advances in Neural Information Processing Systems, vol. 8, pp. 1024–1030 (1996)
130. Zhang, W.: Algorithms for partially observable Markov decision processes. Ph.D. thesis, Citeseer (2001)
131. Zhou, J., Troyanskaya, O.G.: Predicting effects of noncoding variants with deep learning-based sequence model. Nat. Methods **12**(10), 931–934 (2015)