Notes on Reinforcement Learning

Paulo Eduardo Rauber

2014

1 Introduction

Reinforcement learning is the study of agents that act in an environment with the goal of maximizing cumulative reward signals. The agent is not told which actions to take, but discovers which actions yield most rewards by trying them. Its actions also may affect not only the immediate rewards but rewards for the next situations. There are several methods to solve the reinforcement learning problem.

One of the challenges in reinforcement learning is the exploration versus exploitation problem, which is the trade-off between obtaining rewards from perceived safe options against exploring other possibilities which may be advantageous.

A reinforcement learning problem can be divided into four subelements: policy, reward function, value function and a model.

A policy specifies how an agent behaves in a given scenario. It can be seen as a mapping from the perceived environmental state to actions.

A reward function defines the goal in a reinforcement learning problem. It maps each state of the environment to a numerical reward, indicating the intrinsic desirability of that state. An agent's sole objective is to maximize the total reward it receives on the long run. The reward function, therefore, should not be alterable by the agent.

A value function maps a state to an estimate of the total reward an agent can expect to accumulate over the future starting from that state. Even though a given state may have a small reward, it may be a prerequisite for achieving states with higher rewards.

A model is an optional element of a reinforcement learning system. A model describes the behavior of the environment. For example, it can be used to predict the results of actions (both states and rewards). It is possible to have agents that learn by trial and error, construct a model of the environment and use the model for planning.

2 Evaluative Feedback

The n-armed bandit problem is very useful to introduce a number of basic learning methods. Suppose an agent is faced with a choice among n different actions. After each action, a numerical reward is sampled based on a stationary probability distribution associated to that particular action. The agent's goal is to maximize the total reward after a number of iterations.

In this case, the value of an action is the expected reward that follows it. If an agent knew the value of every action, it would suffice to choose the action with the highest value.

The agent can maintain an estimate to the value of each action. The action with the highest estimated value is called greedy. Choosing the greedy action (exploitation) may guarantee higher rewards in the short run, while choosing non-greedy actions may lead to better outcomes later.

Let $Q_t(a)$ denote the estimated value action a at iteration t. We also let $Q^*(a)$ denote the value of a. A natural way to estimate the value of action a is to average the rewards already received. Concretely, if k_a denotes the number of times action a was chosen, we can compute $Q_t(a)$ as:

$$Q_t(a) = \frac{r_1 + \ldots + r_{k_a}}{k_a}.$$

By the law of large numbers, when $k_a \to \infty$, $Q_t(a) \to Q^*(a)$.

A very simple alternative to always choosing the action with the highest estimated value is the ϵ-greedy selection rule. This rule chooses the greedy action with probability $1 - \epsilon$ and a random action (uniformly) with probability ϵ. Using this rule, it is extremely important that ties be broken at random. After infinite iterations, every action will be sampled infinite times. Therefore, $Q_t(a)$ converges to $Q^*(a)$ and the chance of choosing the optimal action converges to $1 - \epsilon$.

Another possibility is to weight the probability of choosing an action with its current value estimate. Let $\pi_t(a)$ denote the probability of choosing action a at iteration t and n denote the total number of actions. The following equation is called the softmax action selection rule:

$$\pi_t(a) = \frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}}.$$

In this case, the probability of an action being selected is in proportion to its estimated value. The parameter τ is called temperature. When τ is large, the probabilities are more uniform. When τ is near zero, the rule is greedier. In the case of two actions, this function becomes the sigmoidal function.

Fixing an action a and letting k denote the number of times it has been chosen, the equation given for estimating action values as an average of observed rewards can be rewritten as:

$$Q_{k+1} = Q_k + \frac{1}{k+1}(r_{k+1} - Q_k).$$

If the problem is non-stationary, however, it may be better to estimate the value of an action as:

$$Q_{k+1} = Q_k + \alpha(r_{k+1} - Q_k).$$

Where $0 < \alpha \leq 1$ is called the step-size parameter. In this case, it can be shown that the value of Q_k is given by:

$$Q_k = (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i.$$

Since it can be shown that $(1-\alpha)^k + \sum_{i=1}^{k} \alpha(1-\alpha)^k = 1$, this update rule is a weighted average of the initial estimate and the observed rewards. The recently observed rewards have more importance than older rewards. In fact, the weight decreases exponentially for older rewards.

If we denote by $\alpha_k(a)$ the step-size parameter used to process the reward given after the k-th selection of action a, the choice of $\alpha_k(a) = \frac{1}{k}$ leads to the sample average method, while $\alpha_k(a) = \alpha$ leads to the constant step-size method.

There are two conditions that guarantee the convergence of the value estimate to the value of an action a:

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty,$$

and

$$\sum_{k=1}^{\infty} \alpha_k^2(a) < \infty.$$

Therefore, convergence is guaranteed for the sample average method, but not for the constant step size method. However, constant step sizes are advantageous if the problem is non-stationary, meaning that the value of actions may change during the iterations.

There is a simple trick that can be employed to make an agent explore in early iterations. If the initial value estimates are set optimistically, i.e., significantly higher than the value of the optimal action, even a greedy agent will explore considerably in the initial iterations until the value estimates are lowered.

Suppose that an observed variable (the state of the system) contained information about the next reward for a given action. It would be possible to achieve better results than using the strategies described in this chapter. This motivates the full reinforcement learning problem.

3 The Reinforcement Learning Problem

A reinforcement learning agent interacts with the environment at each of a sequence of discrete time steps $t = 0, 1, 3, \ldots$. At each time step t, the agent receives some representation of the environment's state $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states. The agent then selects an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of available actions in state s_t. One time step later, the agent receives a reward r_{t+1} and a new state s_{t+1}. The agent implements a policy $\pi_t(s, a)$: the probability that $a_t = a$ given that $s_t = s$.

Modeling a problem as a reinforcement learning problem is challenging. Particularly, the boundary between agent and environment is sometimes not clear. A good strategy may be to abstract as much of the problem as

possible and let the agent focus on learning that which is hard to explicitly program. For instance, the action primitives for a robot may be pre-programmed tasks instead of mechanical control. Anything that cannot be controlled directly by the agent should be considered part of the environment, including rewards.

At each time step, the agent receives a reward $r_t \in \mathbb{R}$. The objective of the agent is to maximize the total amount of reward it receives on the long run. The choice of a reward function is critical in a reinforcement learning problem. It is particularly important not to impart previous knowledge into the reward function about how to solve a problem, but only what the agent should achieve. Previous knowledge should be imparted into state value estimates, which will be discussed later.

Let $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$ denote the sequence of rewards received after time step t. A function R_t from a sequence of rewards to a real number is called a return function, and its value for a specific sequence is called return. A reinforcement learning agent maximizes the expected return.

In a simple formulation, the return function could be defined as $R_t = r_{t+1} + r_{t+2} + \ldots + r_T$, where T is the final time step. When T is finite, we call the problem episodic. An episode ends when the agent reaches a terminal state. After that, the time is reset and the agent samples from a distribution of initial states.

However, when $T = \infty$, the simple formulation of R_t may be problematic, since the return could be infinite. This is called a continuous problem. In this case, it is better to consider the discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \tag{1}$$

where $0 \leq \gamma < 1$ is called the discount rate. The discount rate determines the present value of future rewards. A reward received k time steps into the future is only worth γ^{k-1} times what it would be worth if it were received immediately. In this formulation, as long as the sequence $\{r_k\}$ is bounded, the (discounted) return is finite. The choice of γ is important to determine how future-oriented is the agent.

In both continuous and episodic problems, the expected return for each state can be estimated by averaging the returns observed in several instances of the problem (which may be approximations in the continuous case).

The choice of a reward function is dependent on the definition of return. For instance, consider an agent which tries to escape a maze. In the episodic case, the reward could be -1 for every move the agent makes until it reaches the exit. In this case, after T time steps, the reward for the agent would be $-T$, and maximizing the expected return would imply in finding the exit early. In the discounted case, the reward could be 1 whenever the agent found the final step and 0 otherwise. In this case, the expected return would be γ^k, and minimizing k would be implied. However, if the same formulation were used in the episodic, undiscounted case, the agent would have no incentive to find the exit early.

An episodic problem can be modeled as a continuous problem by simply creating a state which yields no rewards and transitions only to itself. In fact, if we let $\gamma = 1$, we can use the same formulation given in Eq. 1 for continuous tasks, as long as $T = \infty$ and $\gamma = 1$ are never both true.

In a reinforcement learning problem the state represents whatever information is available to the agent. States may be immediate sensations or highly processed versions of data and are often represented by vectors. States don't need to represent the environment perfectly or even all that could be useful in making marginally better decisions.

However, a state should respect the Markov property as well as possible. A state that summarizes all that is relevant for decision making that was observed in past states is said to have the Markov property. More precisely, given the sequence of states, action and rewards $s_t, a_t, r_t, \ldots r_1, s_0, a_0$ (called history) observed by an agent, the state signal is said to have the Markov property if:

$$P(\mathbf{s}_{t+1} = s', \mathbf{r}_{t+1} = r' | s_t, a_t, r_t, \ldots, r_1, s_0, a_0) = P(\mathbf{s}_{t+1} = s', \mathbf{r}_{t+1} = r' | s_t, a_t),$$

for all s', r' and histories. We call the probability distribution function on the right side the one-step dynamics of the reinforcement learning problem. It can be shown that the one-step dynamics can be used to derive the probability of all future states and expected rewards only from the initial state.

A reinforcement learning problem that satisfies the Markov property is called a Markov decision process, often called *MDP*. In particular, if the state and action sets are finite, the problem is called a finite MDP.

A Markov decision process is defined by its state and action sets and by the one-step dynamics of the environment. Given any state s and action a, the probability $\mathcal{P}^a_{ss'}$ of each next state s' is given by:

$$\mathcal{P}^a_{ss'} = P(\mathbf{s}_{t+1} = s' | \mathbf{s}_t = s, \mathbf{a}_t = a) = \sum_{r'} P(\mathbf{s}_{t+1} = s', \mathbf{r}_{t+1} = r' | \mathbf{s}_t = s, \mathbf{a}_t = a).$$

Given any state s, action a and next state s', the expected value of the next reward is given by:

$$\mathcal{R}^a_{ss'} = \mathbb{E}[\mathbf{r}_{t+1}|\mathbf{s}_t = s, \mathbf{a}_t = a, \mathbf{s}_{t+1} = s'] = \frac{1}{\mathcal{P}^a_{ss'}} \sum_{r'} r' P(\mathbf{s}_{t+1} = s', \mathbf{r}_{t+1} = r'|\mathbf{a}_t = a, \mathbf{s}_t = s).$$

The value $V^\pi(s)$ of a state s is defined as the expected return of starting in state s and following the policy π. The policy assigns a probability $\pi(s,a)$ to taking an action a when in a state s. More precisely:

$$V^\pi(s) = \mathbb{E}_\pi[\mathbf{R}_t|\mathbf{s}_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k \mathbf{r}_{t+k+1}|\mathbf{s}_t = s].$$

Similarly, the value of taking an action a when in state s and afterwards following the policy π is denoted by $Q^\pi(s,a)$ and defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi[\mathbf{R}_t|\mathbf{s}_t = s, \mathbf{a}_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k \mathbf{r}_{t+k+1}|\mathbf{s}_t = s, \mathbf{a}_t = a].$$

The function V^π is called state value function for policy π and the function Q^π the action value function for policy π. These two functions can be estimated, for a given policy, by observing several instances of the reinforcement learning problem.

Given a policy, another way of stating the state and action value functions is:

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')], \tag{2}$$

and

$$Q^\pi(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')].$$

These two equations are called the Bellman equations and can be interpreted intuitively. Note that, given a policy and the one-step dynamics of the problem, each of the equations defines a system of $|\mathcal{S}|$ linear equations and variables. For any given policy, it is possible to find the *unique* expected return for any given state and action at a state.

It is possible to prove, in the case of a continuous discounted return problem, that adding a constant to all the rewards only adds another constant to the value of all states. However, in the case of a episodic task, this is not generally true, due to the variable number of time steps that may occur in each episode.

There are two alternative ways of writing the state and action value functions in terms of each other:

$$V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s,a),$$

and

$$Q^\pi(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')].$$

Solving a reinforcement learning problem consists in finding a policy that has high expected return. Let $\pi > \pi'$ if and only if $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$. The optimal policy π^* is such that $\pi^* \geq \pi$ for any policy π. It is possible to show that this policy exists but is not necessarily unique. We define the optimal state value function V^* and the optimal action value function Q^* as:

$$V^*(s) = \max_\pi V^\pi(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')],$$

and

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')],$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. These are called the Bellman optimality equations. Each equation gives a system of nonlinear equations that can be used to find $V^*(s)$ or $Q^*(s,a)$ for any state s and action a. In practice, however, solving nonlinear systems of equations in problems with many states may be unfeasible.

An optimal policy π^* can be found given V^* or Q^*. In the case of V^*, it suffices to choose one of the actions a that maximizes the right hand side of its Bellman optimality equation with uniform probability. In the case of Q^*, it suffices to choose uniformly the a such that $Q^*(s, a)$ is maximal.

A large amount of memory may be required to build up approximations of value functions, policies and models. Therefore, in many practical problems, these functions must be approximated by a parameterized function.

4 Dynamic Programming

The term dynamic programming refers to a collection of reinforcement learning algorithms that can be used to compute optimal policies given a perfect model of the environment (one step dynamics). It is assumed that the set of states and actions are finite.

Policy evaluation is the name given to computing the state value function V^π given an arbitrary policy π. As already stated, it is possible to solve the system of linear equations with variables $V^\pi(s)$ (Eq. 2). However, it is also possible to create a sequence of approximations to V^π by the following rule:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s)],$$

for all $s \in \mathcal{S}$. This sequence is guaranteed to converge to $V^\pi(s)$ for all $s \in \mathcal{S}$, since $V_k = V_{k+1}$ for some k implies on the solution to the Bellman equations. The initial value for each state can be arbitrary. This method is called iterative policy evaluation.

In a given iteration of policy evaluation, instead of computing the new estimate for each state using the estimate from the previous iteration, it is also valid to consider the latest estimate available for each state. This variation is classified as an in-place algorithm.

A deterministic policy π is one such that, for all $s \in \mathcal{S}$, $\pi(s,a) = 1$ for some a and $\pi(s,b) = 0$ for all $b \neq a$. Thus, in the case of deterministic policies, it is simpler to consider π as a function from states to actions.

Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$, $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$. The policy improvement theorem guarantees that $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in \mathcal{S}$. This gives a natural way of improving a given policy π to a better policy π' by the following rule:

$$\pi'(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s)].$$

A sequence of evaluations and improvements can be applied to an arbitrary initial policy. Once a policy can no longer be improved, it is guaranteed to be optimal by the Bellman optimality equations. This technique for finding an optimal policy is called policy iteration.

A very simple and efficient alternative to policy iteration is called value iteration. This technique successively improves the estimates for the value of each state $s \in \mathcal{S}$, beginning with an arbitrary value $V_0(s)$, by the following rule:

$$V_{k+1} = \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')].$$

It can be shown that the sequence $\{V_k\}$ converges to V^*. Algorithm 4 describes how an optimal policy can be obtained by value iteration.

Algorithm 1 Value iteration

Input: set of states $\mathcal{S}$, one step dynamics functions $\mathcal{P}$ and $\mathcal{R}$, discount factor γ.
Output: optimal deterministic policy π.

```
for each s ∈ S do
    V(s) ← 0
end for
repeat
    Δ ← 0
    for each s ∈ S do
        v ← V(s)
        V(s) ← max_a Σ_{s'} P^a_{ss'}[R^a_{ss'} + γV(s')]
        Δ ← max(Δ, |v − V(s)|)
    end for
until Δ < θ
for each s ∈ S do
    π(s) = arg max_a Σ_{s'} P^a_{ss'}[R^a_{ss'} + γV(s)]
end for
```

Asynchronous algorithms can also be used to update the estimates for V.

There are two main disadvantages of both policy iteration and value iteration: they required expensive sweeps over the state space, which may be prohibitively large, and a complete model of the environment.

5 Monte Carlo Methods

Monte Carlo methods can be used to solve the reinforcement learning problem without a model (one-step dynamics) for the environment. The agent learns by averaging observed returns. Even if it is possible to construct a model of the environment, it is often easier to let the agent learn in this way.

These methods also work by successive policy evaluation and improvement. The most simple way to estimate the value of a state is to average the observed returns after a number of episodes in which the state appears. If only the first occurrence of a state s in an episode is considered to improve the estimate for $V(s)$, then the method is called first-visit. Otherwise, when every occurrence of a state s is considered, the method is called every-visit. In both cases, when the number of visits to s tends to infinity, this kind of estimate tends to $V(s)$.

Since a model is not available for Monte Carlo methods, it is useful to estimate action values instead of state values. Evaluating a policy π by a Monte Carlo method consists on experiencing several episode and averaging the returns that follow every possible state action pair (s, a) to obtain an estimate for $Q(s, a)$.

As in the previous section, we are interested in improving a policy π. In this case, we do this episode-by-episode. However, unlike in the previous section, the policies must be ϵ-soft, since we need to guarantee that the method improves its estimate for the action values. Algorithm 5 illustrates a Monte Carlo based algorithm to solve the reinforcement learning problem. It is assumed that there is no discounting, since the length of each episode *should* be finite.

Algorithm 2 Monte Carlo control algorithm

Input: set of states $\mathcal{S}$, number of episodes N, probability of choosing random action ϵ.
Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
    for each action a ∈ A(s) do
        Q(s, a) ← 0
        n(s, a) ← 0
    end for
end for
for each i in {1, ..., N} do
    Experience a new episode e following an ε-greedy policy based on Q.
    for each state-action pair (s, a) in the episode e do
        R ← return following (s, a) in the episode e.
        n(s, a) ← n(s, a) + 1
        Q(s, a) ← Q(s, a) + 1/n(s,a) [R − Q(s, a)]
    end for
end for
for each state s ∈ S do
    π(s) ← arg max_a Q(s, a)
end for
```

A method to solve the reinforcement learning problem that uses the value of other states to compute the value of a state s is said to bootstrap. Since Monte Carlo methods do not bootstrap, in contrast with dynamic programming, it is possible to evaluate the value of each state separately, which can be necessary in some situations. It is also possible to evaluate a policy while following another, which is described in detail in [1].

6 Temporal-Difference Learning

Temporal-difference learning methods are also capable of learning from experience without a model of the environment. However, differently from Monte Carlo methods, they bootstrap. Intuitively, this allows them to extract more information from their experiences.

Instead of waiting for the end of an episode to update their state or action values, a TD agent updates at every time step. The simplest TD method is called $TD(0)$, and updates the state value estimates using the following rule:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \tag{3}$$

where α is the learning rate and γ is a discount parameter. This rule updates the value of a state based on the reward after one time-step and the estimated value of the next state.

Sarsa is an on-policy method to solve the control problem: finding an optimal policy by interacting with the environment. The idea is to adapt Equation 3 to estimate action values, which involves using the tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, which gives the method's name. As with other control algorithms, the starting point is an arbitrary ϵ-soft policy, which is used to estimate action values and improve the policy, which is done after each time step. Sarsa is exemplified by algorithm 6, which uses ϵ-greedy policies.

Algorithm 3 Sarsa control algorithm

Input: set of states $\mathcal{S}$, number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ.

Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
    for each action a ∈ A(s) do
        Q(s, a) ← 0
    end for
end for
for each i in {1, ..., N} do
    s ← initial state for episode i
    Select action a for state s according to an ε-greedy policy based on Q.
    while state s is not terminal do
        r ← observed reward for action a at state s
        s' ← observed next state for action a at state s
        Select action a' for state s' according to an ε-greedy policy based on Q.
        Q(s, a) ← Q(s, a) + α[r + γQ(s', a') − Q(s, a)]
        s ← s'
        a ← a'
    end while
end for
for each state s ∈ S do
    π(s) ← arg max_a Q(s, a)
end for
```

Another advantage of TD methods over Monte Carlo methods is that episodes don't need to be finite. Also, while Monte Carlo methods minimized the mean squared-error on the training set, TD methods find the maximum likelihood model for the underlying Markov process, which achieves better generalization. Finding the maximum likelihood model requires that a set of episodes be experienced repeatedly by the algorithms until convergence of the action value function Q, in a process that is called batch-updating.

An alternative to Sarsa is Q-Learning, an off-policy control method. This means that Q-learning learns the action values for a policy that it does not use to act on the environment. The main difference to Sarsa is the update rule, which takes the form:

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

Instead of choosing a' using the current policy, which may be ϵ-soft for learning purposes, this update rule considers the best possible action according to the current estimate. This avoids the problem of misattributing an inherent low value for a state in which a bad action choice was made. For the same reason, Q-learning may

converge faster but have lower efficacy during the learning phase (called online efficacy). Algorithm 6 exemplifies a particular implementation of Q-learning.

Algorithm 4 Q-learning control algorithm

Input: set of states $\mathcal{S}$, number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ.

Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
    for each action a ∈ A(s) do
        Q(s, a) ← 0
    end for
end for
for each i in {1, ..., N} do
    s ← initial state for episode i
    while state s is not terminal do
        Select action a for state s according to an ε-greedy policy based on Q.
        r ← observed reward for action a at state s
        s' ← observed next state for action a at state s
        Q(s, a) ← Q(s, a) + α[r + γ max_{a'∈A(s')} Q(s', a') − Q(s, a)]
        s ← s'
    end while
end for
for each state s ∈ S do
    π(s) ← arg max_a Q(s, a)
end for
```

Alternatively, the update rule for Q-learning can be taken as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \sum_a \pi(s_t, a) Q(s_{t+1}, a) - Q(s_t, a_t)].$$

In this case, the probability of choosing each action is taken into account, which may improve its online efficacy.

It is often possible to design a more effective learning algorithm when some part of the environment dynamics is known. For instance, if the state following a state action pair is always known, it is possible to base policies on the value of possible successor states, by learning what is called an afterstate value function. In that case, the control algorithm can learn the state value function V instead of Q.

7 Eligibility Traces

The n-step return at time t is defined as:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}).$$

This return can be used to update state (action) values:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t^{(n)} - V(s_t)].$$

When $n = 1$, the update rule above is equivalent to the temporal difference learning rule. When $n \to \infty$, this rule is equivalent to the rule used in Monte Carlo methods.

It is also possible to average $R_t^{(i)}$ for different i. One particularly important weighted average of variable step returns is R_t^λ, called λ-return, which is defined by:

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}.$$

In this way, the weights given to each n-step return sum to 1. The 1-step return has the highest weight, which decreases by a factor of λ for each following return. When $\lambda = 0$, the definition is once again equivalent to temporal difference learning with one step return. When $\lambda \to 1$, the method is equivalent to Monte Carlo. This gives a somewhat simple way to control the balance between state (action) value estimates and looking ahead for rewards. This idea has been shown to be very effective in practice for good choices of λ.

In practice, the update rule based on R_t^λ is efficiently implemented by using eligibility traces. The eligibility trace e_t for a state s at time t is defined as:

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

The following update rule should be applied to *every* state (action) value:

$$V(s) \leftarrow V(s) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]e_t(s). \tag{4}$$

Intuitively, the observed error for the estimate of the value of the current state s_t changes the estimate of s by a factor of $\alpha e_t(s)$, which is proportional to how soon state s was last visited. In this context, λ is called the trace decay factor.

The control algorithms presented in the previous section can be adapted to use eligibility traces as described by 7 and 7. Note that the algorithm $Q(\lambda)$ needs to erase its eligibility traces once a non-greedy action is taken because of its off-policy evaluation.

The eligibility traces presented in this section are also called accumulating traces. Replacing traces are an alternative definition that may be better suited to some applications, particularly when the agent is likely to take the same action at a given state several times, causing a detrimental accumulation of its eligibility trace. The replacing trace e_t for a state s at time t is defined as:

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

Algorithm 5 Sarsa(λ) control algorithm

Input: set of states $\mathcal{S}$, number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ, trace decay λ

Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
    for each action a ∈ A(s) do
        Q(s, a) ← 0
        e(s, a) ← 0
    end for
end for
for each i in {1, ..., N} do
    Set the eligibility trace e to zero for every valid state action pair.
    s ← initial state for episode i
    Select action a for state s according to an ε-greedy policy based on Q.
    while state s is not terminal do
        r ← observed reward for action a at state s
        s' ← observed next state for action a at state s
        Select action a' for state s' according to an ε-greedy policy based on Q.
        e(s, a) ← e(s, a) + 1
        for each s'' ∈ S do
            for each action a'' ∈ A(s'') do
                Q(s'', a'') ← Q(s'', a'') + α[r + γQ(s', a') − Q(s, a)]e(s'', a'')
                e(s'', a'') ← γλe(s'', a'')
            end for
        end for
        s ← s'
        a ← a'
    end while
end for
for each state s ∈ S do
    π(s) ← arg max_a Q(s, a)
end for
```

Algorithm 6 Q(λ) control algorithm

Input: set of states $\mathcal{S}$, number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ, trace decay λ

Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
  for each action a ∈ A(s) do
    Q(s, a) ← 0
    e(s, a) ← 0
  end for
end for
for each i in {1, ..., N} do
  Set the eligibility trace e to zero for every valid state action pair.
  s ← initial state for episode i
  Select action a for state s according to an ε-greedy policy based on Q.
  while state s is not terminal do
    r ← observed reward for action a at state s
    s' ← observed next state for action a at state s
    Select action a' for state s' according to an ε-greedy policy based on Q.
    Select the greedy action a* for state s' according to Q, preferring a' when tied.
    e(s, a) ← e(s, a) + 1
    for each s'' ∈ S do
      for each action a'' ∈ A(s'') do
        Q(s'', a'') ← Q(s'', a'') + α[r + γQ(s', a*) − Q(s, a)]e(s'', a'')
        if a* = a' then
          e(s'', a'') ← γλe(s'', a'')
        else
          e(s'', a'') ← 0
        end if
      end for
    end for
    s ← s'
    a ← a'
  end while
end for
for each state s ∈ S do
  π(s) ← arg max_a Q(s, a)
end for
```

8 Generalization and Function Approximation

There are situations in which it is inadvisable to store state or action values in arrays. In large state spaces, some spaces may be seen rarely. In these cases, the state (action) values must be generalized across states. The kind of generalization discussed in this section is based on function approximation, which is studied extensively in machine learning.

The state value function (at time t) V_t can be approximated by a function f such that $f(\vec{\theta}_t, s_t) = V_t(s_t)$, where $\vec{\theta}_t$ is a vector in $\mathbb{R}^m$ and s_t is a state. In this case, the value $V(s_t)$ should depend only on the parameters $\vec{\theta}_t$, and the problem is then formulated as finding the best $\vec{\theta}_t$ to *fit* the real value function. In this case, changing the value of $\vec{\theta}_t$ typically changes the value estimates for several states.

Several methods already presented shifted the value $V_t(s_t)$ of a state s_t towards a new value $V_{t+1}(s_t)$. In the case of function approximation, the new value may be considered as new sample for supervised learning. Concretely, the approximate value function V_t can be learned from pairs (s_i, v_i), for $i \leq t$, where s_i is a state and v_i is its observed value. In the case of $TD(\lambda)$, for instance, $v_i = R_i^\lambda$.

Using this formulation, any regression method may be used to approximate V_t. This section will focus on linear regression by gradient descent, mainly due to its simplicity.

A common performance measure for evaluating function approximation is the mean squared error:

$$\text{MSE}(\vec{\theta}_t) = \sum_{s \in S} P(s)[V^\pi(s) - V_t(s)]^2.$$

In the equation above, S is the set of states, P is a probability distribution over states, $V^\pi(s)$ is the value of state s under the policy π and $V_t(s) = f(\vec{\theta}_t, s)$ is the current value estimate for state s. In most cases, the distribution P should describe the frequency with which each state is encountered while the agent interacts with the environment under π.

A vector $\vec{\theta}^*$ such that $\text{MSE}(\vec{\theta}^*) \leq \text{MSE}(\vec{\theta})$ for every vector $\vec{\theta}$ is called a global optimum. Finding a global optimum is rarely practical for complex functions. Several methods used in practice are guaranteed only to find local optima, which are global optima in a restricted domain of the function.

Gradient descent is a very widespread technique for finding local optima in smooth functions. Stochastic gradient descent is a variation that is particularly useful in reinforcement learning. Given an estimate for the parameter vector $\vec{\theta}_t$ and a current state s_t, the estimate may be updated in the following way:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}_t}[V^\pi(s_t) - V_t(s_t)]^2.$$

In the equation above, α is the learning rate and $\nabla_{\vec{\theta}_t}[V^\pi(s_t) - V_t(s_t)]^2$ is the gradient of $[V^\pi(s_t) - V_t(s_t)]^2$ with respect to $\vec{\theta}_t$. This gradient is the vector of partial derivatives of the error term (the squared term) with respect to each component of $\vec{\theta}_t$. Intuitively, at each time step, the estimate is changed in the direction that minimizes the error (squared term) observed for state s_t. The direction in which the error is lowered for each component of $\vec{\theta}_t$, for a particular state s_t, is given by this gradient.

The equation above can be rewritten in the following way:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[V^\pi(s_t) - V_t(s_t)]\nabla_{\vec{\theta}_t} V_t(s_t).$$

If α decays with time in the same way as described in section 2, this method is guaranteed to converge to a local optimum in a finite number of steps.

Naturally, if $V^\pi(s)$ were available for all observed states s, there would be no need for function approximation. Therefore, in practice, an estimate of $V^\pi(s_t)$, denoted v_t, should be used to update the current parameter vector:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[v_t - V_t(s_t)]\nabla_{\vec{\theta}_t} V_t(s_t).$$

Suppose the states are sampled according to policy π. Let R_t denote the return following each state s_t. Since the expected value of R_t is the true value of s_t, R_t is called an unbiased estimate and gradient descent is guaranteed to converge to a locally optimum approximation of $V^\pi(s_t)$ when $v_t = R_t$. Therefore, Monte Carlo policy evaluation has the same guarantee. Unfortunately, the same guarantee does not apply to methods based on $TD(\lambda)$ for $\lambda < 1$. However, in practice, temporal difference methods are often more successful in policy evaluation.

The updating of $\vec{\theta}_t$ using $TD(\lambda)$ can be formulated in the following way:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[R_t^\lambda - V_t(s_t)]\nabla_{\vec{\theta}_t} V_t(s_t).$$

It can be shown that this is equivalent to the following update rule using eligibility traces:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t,$$

with

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

and

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t),$$

and $\vec{e}_0 = \vec{0}$.

In this case, there is a vector of eligibility traces with the same number of components as $\vec{\theta}_t$.

Algorithm 7 is a complete algorithm for approximating V^π using gradient descent. The function V is, implicitly, a function of $\vec{\theta}$.

Algorithm 7 Gradient descent TD(λ) value estimation algorithm

Input: policy π, number of episodes N, learning rate α, discount factor γ, trace decay λ
Output: parameter vector $\vec{\theta}$

```
θ⃗ ← 0⃗
for each i in {1, ..., N} do
    e⃗ ← 0⃗
    s ← initial state for episode i
    while state s is not terminal do
        a ← π(s)
        r ← observed reward for action a at state s
        s′ ← observed next state for action a at state s
        δ ← r + γV(s′) − V(s)
        e⃗ ← γλe⃗ + ∇_θ⃗ V(s)
        θ⃗ ← θ⃗ + αδe⃗
        s ← s′
    end while
end for
```

Linear functions on the parameter vector $\vec{\theta}_t$ are widely used for function approximation mainly due to their simplicity. In this case, a state is often represented by a *feature vector* $\vec{\phi}_s$, which should be useful for generalization across states. Given a parameter vector $\vec{\theta}_t$, the value $V_t(s)$ can be computed as:

$$V_t(s) = f(\vec{\theta}_t, s) = \sum_i \theta_t(i)\phi_s(i).$$

In this case, it is clear that $\nabla_{\vec{\theta}_t} V_t(s) = \vec{\phi}(s)$.

In the case of linear function approximation, there is only one local optimum and, therefore, only one global optimum

The choice of feature vectors is crucial to the success of any function approximation method. Intuitively, the features should correspond to natural features of the state, which should facilitate generalization. For instance, a mobile robot might represent its state as a combination of its position, velocity, remaining power etc. In the case of linear models, it might be necessary to create features that are the combinations of other features, since linear models are, in general, incapable of modeling interactions such as feature i being good only on the absence of feature j.

High dimensional state spaces often need to be simplified for use with linear models. There are several techniques to accomplish this. The states can be represented in feature vectors by discretizing the state space into overlapping hyperspheres (coarse coding) or by tiling into hypercubes (tile coding, which may combine several tilings with different offsets). States can also be represented by feature vectors of distance calculations to random points sampled in the state space (similar to coding by radial basis functions).

The extension of function approximation from value prediction to action value prediction is straightforward. In this case, Q should be approximated by a function f of $\vec{\theta}_t$, s and a. Algorithms 8 and 9 illustrate Sarsa(λ) and Q(λ) based on linear function approximation.

Algorithm 8 Sarsa(λ) control algorithm for linear function approximation

Input: feature vector $\phi_{s,a}$ for all state-action pairs (s, a), number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ, trace decay λ

Output: parameter vector $\vec{\theta}$

```
θ⃗ ← 0⃗
for each i in {1, ..., N} do
    e⃗ ← 0⃗
    s ← initial state for episode i
    for each a ∈ A(s): do
        Q(a) ← Σ_i θ(i)φ_{s,a}(i)
    end for
    a ← arg max_{a∈A(s)} Q(a)
    With probability ε: a ← random action in A(s)
    while state s is not terminal do
        e⃗ ← γλe⃗ + φ⃗_{(s,a)}
        r ← observed reward for action a at state s
        s′ ← observed next state for action a at state s
        δ ← r − Q(a)
        for each a ∈ A(s): do
            Q(a) ← Σ_i θ(i)φ_{s′,a}(i)
        end for
        a′ ← arg max_{a∈A(s′)} Q(a)
        With probability ε: a′ ← random action in A(s′)
        δ ← δ + γQ(a′)
        θ⃗ ← θ⃗ + αδe⃗
        s ← s′
        a ← a′
    end while
end for
```

Algorithm 9 Q(λ) control algorithm for linear function approximation

Input: feature vector $\phi_{s,a}$ for all state-action pairs (s, a), number of episodes N, learning rate α, probability of choosing random action ϵ, discount factor γ, trace decay λ

Output: parameter vector $\vec{\theta}$

```
θ⃗ ← 0⃗
for each i in {1, ..., N} do
    e⃗ ← 0⃗
    s ← initial state for episode i
    for each a ∈ A(s): do
        Q(a) ← Σ_i θ(i)φ_{s,a}(i)
    end for
    while state s is not terminal do
        if  with probability 1 − ε: then
            a ← arg max_a Q(a)
            e⃗ ← γλe⃗ + φ⃗_{(s,a)}
        else
            a ← random action in A(s)
            e⃗ ← φ⃗_{(s,a)}
        end if
        r ← observed reward for action a at state s
        s′ ← observed next state for action a at state s
        δ ← r − Q(a)
        for each a ∈ A(s′): do
            Q(a) ← Σ_i θ(i)φ_{s′,a}(i)
        end for
        a′ ← arg max_{a∈A(s′)} Q(a)
        δ ← δ + γQ(a′)
        θ⃗ ← θ⃗ + αδe⃗
        s ← s′
    end while
end for
```

9 Planning and Learning

Model free reinforcement learning methods, such as temporal difference learning, can be used together with model based methods, such as dynamic programming. In section 3, the one step dynamics of the environment was presented as a complete model, i.e., a mechanism that can be used by the agent to predict the consequence of its actions.

In the case of deterministic environments, where the outcome of an action at a given state is always the same, it is sufficient to store these outcomes in a table as the agent learns. In stochastic environments, it is necessary to learn distribution models that represent the probabilities of each possible outcome and expected rewards.

In this context, planning is the process where the agent simulates, using its internal model, the environment and updates its value estimates.

Algorithm 10 illustrates the combination of direct reinforcement learning, using one-step Q-learning, and model learning in a deterministic environment. This method could be enhanced by eligibility traces, but that was omitted for simplicity.

Algorithm 10 Dyna-Q control algorithm for deterministic environments

Input: set of states $\mathcal{S}$, number of episodes N, number of planning steps per episode K, learning rate α, probability of choosing random action ϵ, discount factor γ.

Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
  for each action a ∈ A(s) do
    Q(s, a) ← 0
  end for
end for
for each i in {1, ..., N} do
  s ← initial state for episode i
  while state s is not terminal do
    Select action a for state s according to an ε-greedy policy based on Q.
    r ← observed reward for action a at state s
    s' ← observed next state for action a at state s
    M(s, a) ← s', r
    Q(s, a) ← Q(s, a) + α[r + γ max_{a'∈A(s')} Q(s', a') − Q(s, a)]
    s_next ← s'
    for each k in {1, ..., K} do
      s ← random previously observed state
      a ← random action previously taken in state s
      s', r ← M(s, a)
      Q(s, a) ← Q(s, a) + α[r + γ max_{a'∈A(s')} Q(s', a') − Q(s, a)]
    end for
    s ← s_next
  end while
end for
for each state s ∈ S do
  π(s) ← arg max_{a∈A(s)} Q(s, a)
end for
```

The main advantage of using model learning is the possibility of updating the value of any state at a given time step. This includes states that are not even visited in a given episode. In practice, this may considerably reduce the necessity of interaction with the environment.

Algorithm 11 presents a variation of the previous algorithm for stochastic environments. The algorithm performs full backup, as in dynamic programming methods, using estimates of $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ learned during interaction with the environment.

However, in the case of non-stationary environments, meaning that the environment changes its behavior with time, a model may become incorrect. In this case, the agent may become stuck behaving suboptimally. Even using ϵ-greedy action selection, some action sequences may become too unlikely. This is a classical instance of the exploitation versus exploration problem. There is a simple heuristic that can overcome this problem in planning agents. If the agent stores the time n since the last execution of action a at state s, the planning step may consider the reward for the state-action combination as $r + \kappa\sqrt{n}$ for a small parameter κ and the reward estimate (given

Algorithm 11 Dyna-Q control algorithm for stochastic environments

Input: set of states $\mathcal{S}$, number of episodes N, number of planning steps per episode K, learning rate α, probability of choosing random action ϵ, discount factor γ.

Output: deterministic policy π, optimal when $N \to \infty$.

```
for each s ∈ S do
    for each action a ∈ A(s) do
        Q(s, a) ← 0
        R(s, s', a) ← 0
        N(s, a) ← 0
        N_n(s, a, s') ← 0
    end for
end for
for each i in {1, ..., N} do
    s ← initial state for episode i
    while state s is not terminal do
        Select action a for state s according to an ε-greedy policy based on Q.
        r ← observed reward for action a at state s
        s' ← observed next state for action a at state s
        N(s, a) ← N(s, a) + 1
        N_n(s, a, s') ← N_n(s, a, s') + 1
        R(s, s', a) ← R(s, s', a) + (r − R(s, s', a)) / N_n(s, a, s')
        Q(s, a) ← Q(s, a) + α[r + γ max_{a'∈A(s')} Q(s', a') − Q(s, a)]
        s_next ← s'
        for each k in {1, ..., K} do
            s ← random previously observed state
            a ← random action previously taken in state s
            Q(s, a) ← Σ_{s'} [N_n(s, a, s') / N(s, a)] [R(s, a, s') + γ max_{a'} Q(s', a')]
        end for
        s ← s_next
    end while
end for
for each state s ∈ S do
    π(s) ← arg max_{a∈A(s)} Q(s, a)
end for
```

by the model) r. In this case, a whole sequence of exploratory actions may be undertaken. In practice, however, controlling κ may be challenging.

In algorithm 10, simulated transitions are sampled uniformly. However, this is usually not the best choice, even for the dynamic programming methods presented in section 4. A simple heuristic called prioritized sweeping may be employed to choose transitions for planning. This heuristic prefers to update values for transitions that go into states that had their values considerably changed in the current episode. A method based on this heuristic, for deterministic environments, is presented in algorithm 12.

Algorithm 12 Prioritized sweeping Dyna-Q control algorithm for deterministic environments

Input: set of states $\mathcal{S}$, number of episodes N, number of planning steps per episode K, learning rate α, probability of choosing random action ϵ, discount factor γ, heap threshold h.

Output: deterministic policy π, optimal when $N \to \infty$.

$\mathcal{H} \leftarrow$ empty maximum priority queue (max-heap)
for each $s \in \mathcal{S}$ **do**
  for each action $a \in \mathcal{A}(s)$ **do**
    $Q(s, a) \leftarrow 0$
  end for
end for
for each i in $\{1, \ldots, N\}$ **do**
  $s \leftarrow$ initial state for episode i
  while state s is not terminal **do**
    Select action a for state s according to an ϵ-greedy policy based on Q.
    $r \leftarrow$ observed reward for action a at state s
    $s' \leftarrow$ observed next state for action a at state s
    $M(s, a) = s', r$
    $p \leftarrow |r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a)|$
    if $p > h$ **then**
      Insert (s, a) into $\mathcal{H}$ with priority p
    end if
    $s_{\text{next}} \leftarrow s'$
    for each k in $\{1, \ldots, K\}$ or empty($\mathcal{H}$) **do**
      $s, a \leftarrow$ first($\mathcal{H}$)
      $s', r \leftarrow M(s, a)$
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a)]$
      for each $\bar{s}, \bar{a}$ predicted to lead to s **do**
        $\bar{r} \leftarrow$ predicted reward for transition $\bar{s}, \bar{a}$
        $p \leftarrow |\bar{r} + \gamma \max_{a \in A(s)} Q(s, a) - Q(\bar{s}, \bar{a})|$
        if $p > h$ **then**
          Insert $(\bar{s}, \bar{a})$ into $\mathcal{H}$ with priority p
        end if
      end for
    end for
    $s \leftarrow s_{\text{next}}$
  end while
end for
for each state $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in A(s)} Q(s, a)$
end for

Algorithm 12 maintains a priority queue (max-heap) with every state-action pair whose estimate would change considerably when updated, ordered by the size of the change. When the pair is updated, the change is propagated to transitions that are predicted to lead to it. This idea dramatically increases the performance of agents in some environments.

Another important heuristic is called trajectory sampling. In this case, transitions are selected for planning following the on-policy distribution. In other words, state-action pairs that are frequently visited are more likely to be sampled for updates by planning.

10 Dimensions of Reinforcement Learning

This section presents a summary of the previous sections.

All the reinforcement learning methods have three ideas in common: estimation of value functions, backing up values and generalized policy iteration.

Value backups may vary in the following dimensions:

Table 1: Variation between backup techniques

	Shallow backups	$\cdots$	Deep backups
Full backups	Dynamic programming	$\cdots$	Exhaustive search
$\cdots$	$\cdots$	$\cdots$	$\cdots$
Sample backups	Temporal difference learning	$\cdots$	Monte Carlo

Another important dimension to be explored in reinforcement learning is function approximation.

In generalized policy iteration, states may be evaluated on-policy (as in Sarsa) or off-policy (as in Q-learning).

There are several important elements to consider while modeling a reinforcement learning problem:

- Return: discounted (continuing) or undiscounted (episodic)
- Value estimation: action value, state value or afterstate value
- Exploration: ϵ-greedy, softmax
- Eligibility traces
- Planning: none, deterministic or stochastic

Some less developed research areas in reinforcement learning include the study of non-Markov problems. An important formulation of the reinforcement learning problem in this scenario is called partially observable Markov decision problem (POMDP). The methods used to solve this problem are computationally expensive and harder to apply for model free agents.

Another important area of research is hierarchical learning, where the agent learns important sequences of action-states as primitive actions.

Teaching reinforcement learning agents is also an important area of study. This involves, for example, building problems of increasing difficulty that an agent must learn in order to achieve its most important goals.

11 Case Studies

Reinforcement learning techniques have been applied extensively to problems in the real world. Framing a real problem in terms of reinforcement learning is often very challenging. Case studies are useful in training the skills necessary to model problems.

In TD-Gammon, for instance, temporal difference learning with eligibility traces has been applied to the game of backgammon with great success. The agent uses multi-layer artificial neural networks as a function approximator and learns by self-training. In competitive games, self-training is the process whereby an agent learns by playing against itself.

Other case studies presented in [1] include the game of checkers, acrobatic robots, dynamic channel allocation and elevator dispatching,

Interestingly, reinforcement learning has also been applied to job shop scheduling using abstract actions on the state space, composed of schedules. This example shows that some useful agents might have to deal with very abstract states and actions.

Challenging decisions in modeling reinforcement learning problems include the quantization of the state space and formulation of the goal in terms of rewards.

License

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

References

[1] Sutton, R.S. and Barto, A.G., *Reinforcement Learning: An Introduction*. MIT Press, 1998.