CSE599i: Online and Adaptive Machine Learning **Winter 2018**

Lecture 17: Reinforcement Learning, Finite Markov Decision Processes

Lecturer: Kevin Jamieson *Scribes: Aida Amini, Kousuke Ariga, James Ferguson, Yujia Wu*

1 Finite Markov decision processes

Finite Markov decision processes (MDPs) [1] [2], are an extension of multi-armed bandit problems. In MDPs, just like bandit problems, we aim at maximizing the total rewards, or equivalently minimizing the regret, by deciding the action every time step. Unlike multi-armed bandit, however, MDPs have states. You can think of MDPs as a set of multi-armed bandit problems, where you are forced to change the bandit machine after every play based on the probabilities that corresponds to each arm. What makes MDPs interesting is that the number of arms and the payout settings are different for each machine. It implies that you should stay at the machine which pays out better to receive better return. Therefore, our basic goal is to estimate the value of each state in terms of the total future reward and identify the actions that get you there.

In this section, we formulate finite MDPs and introduce several solutions based on dynamic programming. Here we assume all the necessary information about the problem is provided, however, as we'll see in later sections, it is not the case in general reinforcement learning problems. We present approaches for these incompletely characterized MDPs in the section 2.

1.1 Multi-armed bandit problem

Let's review the multi-armed bandit problem briefly as a stepping stone to MDPs. The basic premise of the multi-armed bandit problem is that you are repeatedly faced with a choice of k different actions that can be taken. Upon taking an action, you receive a numerical *reward*, R_t, drawn from some distribution based on the action taken, $R_t \sim R(a)$, and are once again given the same choice of k actions. Choosing an action and receiving a reward occurs in a single *time step*. Your objective is to maximize the expected total reward after some fixed number of time steps. Suppose we have a k-armed bandit, and each of the k actions has an underlying *value* or an expected reward, which can be formulated as:

$$Q^*(a) = \mathbb{E}[R_t | A_t = a]$$

where A_t is the action selected on time step t, and R_t is the corresponding reward. If you know the expected value of each action, the planning in multi-armed bandit problem is reduced to selecting the arm with the highest value. Otherwise, you need to estimate the values. Let's denote the estimated value of action a at time step t as $Q_t(a)$. Then, our goal is to estimate the value of each action, i.e. get $Q_t(a)$ close to $Q^*(a)$, while maximizing the total rewards, which makes action selection tricky. On one hand, you want to try all the actions many times to get better estimates, but at the same time, you want to choose actions with high estimates to get better total rewards. This is known as the exploration-exploitation trade-off.

We have already learned major algorithms for the multi-armed bandit problems such as UCB1 [3] and Thompson sampling [4]. Refer to previous lecture notes for a general introduction of stochastic multi-armed bandit problems and their algorithms.

1.2 MDPs formulation

We begin with some definitions and notations to formulate MDPs. The definitions are retrieved from [5] with some modifications in terms of notations and phrasing.

Definition 1. *(Finite Markov decision processes) A finite Markov decision process is defined as a triplet $\mathcal{M}^D = (\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$, where $\mathcal{X}$ is the finite non-empty set of states, $\mathcal{A}$ is the finite non-empty set of actions, and $\mathcal{P}_0$ is the transition probability kernel.*

Definition 2. *(Transition probability kernel) The transition probability kernel $\mathcal{P}_0$ assigns each state-action pair $(x, a) \in \mathcal{X} \times \mathcal{A}$ a probability measure over the next state and the reward $\mathcal{X} \times \mathcal{R}$, where $\mathcal{R} \subset \mathbb{R}$ is bounded by some constant. We shall denote the probability measure by $\mathcal{P}_0(\cdot|x, a)$.*

Definition 3. *(Immediate reward function) The immediate reward function $r : \mathcal{X} \times \mathcal{A} \to \mathcal{R}$ gives the expected reward received when action $a \in \mathcal{A}$ is chosen in state $x \in \mathcal{X}$ regardless of the resulting state. Formally,*

$$r(x, a) = \sum_y \int_s s\mathcal{P}_0(y, s|x, a)ds$$

where s is over the reward space $\mathcal{R}$ and y is over the next state space $\mathcal{X}$.

Definition 4. *(Behavior) The decision maker can select its actions at any stage based on the observed history. A rule describing the way the actions are selected is called a behavior.*

Definition 5. *(Return) The return underlying a behavior is defined as some specific function of the reward sequence. In this note, we consider return with some discount rate, γ such that*

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \end{aligned}$$

where $R_t \sim R(x, a)$ is the reward at time-step t, and $0 \leq \gamma < 1$ is the discount rate. The discount rate puts exponentially smaller weights on the future rewards.

With these notations, the discounted finite MDPs can be described as follows.

At every time step $t = 1, 2, ...$:

- Choose action $A_t \in \mathcal{A}$.
- Transition state as $(X_{t+1}, R_{t+1}) \sim \mathcal{P}_0(\cdot|X_t, A_t)$.
- Observe (X_{t+1}, R_{t+1}).

Objective: Maximize the expected return.

1.3 Policies and value functions

Given the transition probability kernel, a naive way of finding the optimal behavior is to try out all the possible sequences of actions and identify the one that induces the highest return. However, this is generally infeasible because the number of potential sequences increases exponentially with the amount of time steps. A better approach is to limit the search space to *policies*, instead of general behaviors, and use *value functions* to efficiently evaluate them.

Definition 6. *(Stationary policy) Stationary policies are a subset of behaviors where the decision maker considers only the current state to determine the action, and thus, the decision does not depend on the time step. Formally, a policy π maps each state to a probability distribution over actions.*

A stationary policy defines a probability distribution over actions $a \in \mathcal{A}$ for each state $x \in \mathcal{X}$. Let Π be the set of all stationary policies, and from now on we refer to a stationary policy simply as a *policy* for brevity. Fixing player's behavior to a policy π in MDPs gives rise to what is called a Markov reward processes (MRPs). We do not need MRPs in this section, but it will come in handy in section 2.

Definition 7. *(Markov reward processes) Markov reward processes are MDPs where all the decision making is already accounted for by a policy. Formally, an MRP is defined as the pair $\mathcal{M}^R = (\mathcal{X}, \mathcal{P}_0^\pi)$, where*

$$\mathcal{P}_0^\pi(\cdot|x) = \sum_{a \in \mathcal{A}} \mathcal{P}_0(\cdot|x, a)\pi(a|x).$$

We now define two kinds of value functions: state-value function and action-value function.

Definition 8. *(State-value function) The state-value function $V^\pi : \mathcal{X} \to \mathbb{R}$, underlying some fixed policy $\pi \in \Pi$ in an MDP maps a starting state x to the expected return. Formally,*

$$V^\pi(x) = \mathbb{E}_\pi\big[G_t|X_t = x\big], \quad x \in \mathcal{X},$$

where X_t is selected at random such that $\mathbb{P}(X_t = x) > 0$ holds for all states x.

The condition on X_t makes the conditional expectation in $V^\pi(x)$ well defined. Similarly, we can define action-value functions.

Definition 9. *(Action-value function) The action-value function $Q^\pi : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$, underlying a policy $\pi \in \Pi$ in MDP maps a pair of a starting state x and the first action to the expected return. Formally,*

$$Q^\pi(x, a) = \mathbb{E}_\pi\big[G_t|X_t = x, A_t = a\big], \quad x \in \mathcal{X}, a \in \mathcal{A},$$

where X_t and A_t are randomly selected such that $\mathbb{P}(X_t = x) > 0$ and $\mathbb{P}(A_t = a) > 0$ respectively.

A policy is optimal if it achieves the optimal expected return in all states, and it has been proved that there always exists at least one optimal policy [6]. Formally,

$$\exists \pi^* \in \Pi : \pi^* = \arg\max_\pi V^\pi(x), \quad \forall x$$

Optimal value functions map a state or a state-action pair to the expected return that is achieved by following the optimal policy, and they are denoted by V^* and Q^*. The optimal state-value functions and action-value functions are related by the following equations:

$$\begin{aligned} V^*(x) &= \sup_{a \in \mathcal{A}} Q^*(x, a), \quad x \in \mathcal{X} \\ Q^*(x, a) &= r(x, a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, a, y)V^*(y), \quad x \in \mathcal{X}, a \in \mathcal{A}. \end{aligned} \tag{1}$$

where $\mathcal{P}$ is defined as follows.

Definition 10. *(State transition probability kernel) The state transition probability kernel $\mathcal{P}$ gives any triplet $(x, a, y) \in \mathcal{X} \times \mathcal{A} \times \mathcal{X}$ a probability of moving from state x to some other state y provided that action a was chosen in state x. i.e.*

$$\mathcal{P}(x, a, y) = \int_r \mathcal{P}_0(y \times r|x, a)dr.$$

Lastly, there is an important relation between optimal policies and optimal value functions. Notice that any policy $\pi \in \Pi$ which satisfies the following equality is optimal by the definition of value functions.

$$\sum_{a \in \mathcal{A}} \pi(a|x)Q^*(x, a) = V^*(x), \quad x \in \mathcal{X}$$

To have this equation hold, the policy must be concentrated on the set of actions that maximize $Q^*(x, \cdot)$. It means that a policy that deterministically chooses the action that maximizes $Q^*(x, \cdot)$ for all states is optimal. Consequently, the knowledge of the optimal action-value function Q^* alone is sufficient for finding an optimal policy. Besides, by equation 1, the knowledge of the optimal value-function V^* is sufficient to act optimally in MDPs.

Now, the question is how to find V^* or Q^*. If MDPs are completely specified, we can solve them exactly by dynamic programming as we explain in section 1.5. Otherwise, we can predict the dynamics models by estimation methods such as TD and Monte Carlo, which will be described in section 2.1, or even estimate the optimal value function directly as described in section 2.3.

1.4 Bellman equations

A fundamental property of value functions is that they satisfy recursive relationships. Let's begin with a similar and easier example of recursive relationships of returns.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

A similar recursive relationships between the value of the current state and that of the possible successor states hold for any policy π and state x. Fix an MDP $\mathcal{M}^D = (\mathcal{X}, \mathcal{A}, \mathcal{P}_\prime)$, a discount factor γ, and deterministic policy $\pi \in \Pi$. Let r be the immediate reward function of $\mathcal{M}^D$. Then, V^π satisfies the functional equation called the Bellman equation.

$$\begin{aligned} V^\pi(x) &= \mathbb{E}_\pi\left[G_t | X_t = x\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | X_t = x\right] \\ &= r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V^\pi(y). \end{aligned}$$

This Bellman equation is for deterministic policies where $\pi : \mathcal{X} \to \mathcal{A}$, however, you can easily generalize this for stochastic policies by replacing π with a and taking the summation over it. The optimal value function satisfies a specific case of the Bellman equation, namely, the Bellman optimality equation.

$$\begin{aligned} V^*(x) &= r(x, \pi^*(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi^*(x), y) V^*(y) \\ &= \sup_{a \in \mathcal{A}} \left\{ r(x, a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, a, y) V^*(y) \right\}. \end{aligned}$$

1.5 Planning in complete MDPs

Recall that our objective is to maximize the expected return from an MDP, and to this end, we want to know the optimal policy. Also, we can easily obtain the optimal policy once we have found the optimal value function as discussed in the section 1.3. Therefore, if we can learn the optimal value function, we can achieve our goal. In this section, we describe two methods to find the optimal value function V^*: value iteration and policy iteration.

For now, let's assume that we know the transition probability kernel. Then, we can use dynamic programming to solve the Bellman equation for the optimal value function. In fact, Bellman equation is also known as the dynamic programming equation, and it was formulated by Richard Bellman as the necessary condition for optimality associated with dynamic programming [7]. Moreover, the term dynamic programming is generally used as the algorithmic method that nests smaller decision problems inside larger decisions today, however, it was originally invented by Bellman for the very purpose of describing MDPs [8].

Dynamic programming is computationally expensive. Additionally, having knowledge of the complete model is a very strong assumption. Nevertheless, it is an essential foundation for understanding the reinforcement learning algorithms that will be presented in section 2. In the rest of this section, we present value iteration and policy iteration algorithms and analyze them.

1.5.1 Value iteration

We begin with defining an operator.

Definition 11. *(Bellman optimality operator) For any value function $V \in \mathbb{R}^{\mathcal{X}}$, Bellman optimality operator, $T^* : \mathbb{R}^{\mathcal{X}} \to \mathbb{R}^{\mathcal{X}}$ is defined by*

$$(T^*V)(x) = \sup_{a\in\mathcal{A}} \big\{ r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y)V(y) \big\}.$$

If you compare the definition of Bellman optimality equation and Bellman optimality operator, it is obvious that the optimal value function V^* is the solution to

$$V = T^*V.$$

Note that the above equation cannot be solved analytically because T^* contains the sup operation. Instead, value iteration considers the dynamical system $V_{k+1} = T^*V_k$ based on the Bellman optimality operator and value functions, and seeks the optimal value function as the fixed point of the dynamical system, which satisfies the Bellman optimality equation. Once it finds the optimal value function, the optimal policy can be extracted as discussed in the section 1.3.

The algorithm is very simple. We start with an arbitrary V_0 and in each iteration we let

$$V_{k+1} = T^*V_k.$$

Then, the sequence $\{V_k\}_{k=0}^{\infty}$ will converge to V^*. The natural question is why V_k converges to V^* with this algorithm. We prove that the algorithm indeed converges to the unique optimal solution V^*, i.e. the unique fixed point of $V = T^*V$ under the assumption that expected return is calculated over an infinite horizon with some discount. The proof is largely retrieved from [5] and [9].

Definition 12. *(Convergence in norm) Let $V = (V, ||\cdot||)$ be a normed vector space. Let $v_n \in V$ be a sequence of vectors ($n \in \mathbb{N}$). The sequence $(v_n; n \geq 0)$ is said to converge to the vector v in the norm $||\cdot||$ if $\lim_{n\to\infty} ||v_n - v|| = 0$. This will be denoted by $v_n \to_{||\cdot||} v$.*

Note that in a d-dimensional vector space $v_n \to_{||\cdot||} v$ is the same as requiring that for each $1 \leq i \leq d, v_{n,i} \to v_i$ (here $v_{n,i}$ denotes the i-th component of v_n). However, this does not hold for infinite dimensional vector spaces. Take for example $\mathcal{X} = [0,1]$ and the space of bounded functions over $\mathcal{X}$. Let

$$f_n(x) = \begin{cases} 1, & \text{if } x < \frac{1}{n}; \\ 0, & \text{otherwise.} \end{cases}$$

Define f so that $f(x) = 0$ if $x \neq 0$ and $f(0) = 1$. Then $f_n(x) \to f(x)$ for each x (i.e., f_n converges to $f(x)$ pointwise). However, $||f_n - f||_\infty = 1 \not\to 0$.

If we have a sequence of real-numbers $(a_n; n \geq 0)$, we can test if the sequence converges without the knowledge of the limiting value by verifying if it is a Cauchy sequence, i.e., whether $\lim_{n\to\infty} \sup_{m\geq n} |a_n - a_m = 0|$. ('Sequences with vanishing oscillations' is possibly a more descriptive name for Cauchy sequence of reals assumes a limit. The extension of the concept of Cauchy sequences to normed vector spaces is straightforward:

Definition 13. *(Cauchy sequence) Let $(v_n; n \geq 0)$ be a sequence of vectors of a normed vectorspace $V = (V, ||\cdot||)$. Then v_n is called a Cauchy-sequence if $\lim_{n\to\infty} \sup_{m\geq n} ||v_n - v_m|| = 0$.*

Normed vector spaces where all Cauchy sequences are convergent are special: one can find examples of normed vector spaces such that some of the Cauchy sequences in the vector space do not have a limit.

Definition 14. *(Completeness) A normed vector space V is called complete if every Cauchy sequence in V is convergent in the norm of the vector space.*

To pay tribute to Banach, the great Polish mathematician of the first half of the 20th century, we have the following definition.

Definition 15. *(Banach space) A complete, normed vector space is called a Banach space.*

One powerful result in the theory of Banach spaces concerns contraction mappings, or contraction operators. These are special Lipschitzian mappings:

Definition 16. *(L-Lipschitz) Let $V = (V, ||\cdot||)$ be a normed vector space. A mapping $T : V \to V$ is called L-Lipschitz if for any $u, v \in V$,*

$$||Tu - Tv|| \leq L||u - v||.$$

A mapping T is called a non-expansion if it is Lipschitzian with $L \leq 1$. It is called a contraction if it is Lipschitzian with $L < 1$. In this case, L is called the contraction factor of T and T is called an L-contraction.

Note that if T is Lipschitz, it is also continuous in the sense that if $v_n \to_{||\cdot||} v$ then also $Tv_n \to_{||\cdot||} Tv$. This is because $||Tv_n - Tv|| \leq L||v_n - v|| \to 0$ as $n \to \infty$.

Definition 17. *(Fixed point) Let $T : V \to V$ be some mapping. The vector $v \in V$ is called a fixed point of T if $Tv = v$.*

Theorem 1. *(Banach's fixed-point theorem, Contraction mapping) Let V be a Banach space and $T : V \to V$ be a contraction mapping. Then T has a unique fixed point. Further, for any $v_0 \in V$, if $v_{n+1} = Tv_n$ then $v_n \to_{||\cdot||} v$, where v is the unique fixed point of T and the convergence is geometric:*

$$||v_n - v|| \leq \gamma^n ||v_0 - v||.$$

Proof:
Pick any $v_0 \in V$ and define v_n as in the statement of the theorem. We first demonstrate that (v_n) converges to some vector. Then we will show that this vector is a fixed point of T. Finally, we show that T has a single fixed point.

Assume that T is a γ-contraction. To show that (v_n) converges it suffices to show that (v_n) is a Cauchy sequence (since V is a Banach, i.e., complete normed vector-space). We have

$$\begin{aligned}
||v_{n+k} - v_n|| &= ||Tv_{n-1+k} - Tv_{n-1}|| \\
&\leq \gamma||v_{n-1+k} - v_{n-1}|| = \gamma||Tv_{n-2+k} - Tv_{n-2}|| \\
&\leq \gamma^2||v_{n-2+k} - v_{n-2}|| \\
&\vdots \\
&\leq \gamma^n||v_k - v_0||.
\end{aligned}$$

Now,

$$||v_k - v_0|| \leq ||v_k - v_{k-1}|| + ||v_{k-1} - v_{k-2}|| + \cdots + ||v_1 + v_0||$$

and, by the same logic as used before, $||v_i - v_{i-1}|| \leq \gamma^{i-1}||v_1 - v_0||$. Hence,

$$||v_k - v_0|| \leq (\gamma^{k-1} + \gamma^{k-2} + \cdots + 1)||v_1 - v_0|| \leq \frac{1}{1-\gamma}||v_1 - v_0||.$$

Thus,

$$||v_{n+k} - v_n|| \leq \gamma^n(\frac{1}{1-\gamma}||v_1 - v_0||),$$

and so

$$\lim_{n\to\infty} \sup_{k\geq 0} ||v_{n+k} - v_n|| = 0,$$

showing that $(v_n; n \geq 0)$ is indeed a Cauchy sequence. Let v be its limit. Now, let us go back to the definition of the sequence $(v_n; n \geq 0)$:

$$v_{n+1} = Tv_n.$$

Taking the limits of both sides, on the one hand, we get that $v_{n+1} \to_{||\cdot||} v$. On the other hand, $Tv_n \to_{||\cdot||} Tv$, since T is a contraction, hence it is continuous. Thus, the left-hand side converges to v, while the right-hand side converges to Tv, while the left and right-hand sides are equal. Therefore, we must have $v = Tv$, showing that v is a fixed point of T.

Let us consider the problem of uniqueness of the fixed point of T. Let us assume that v, v' are both fixed points of T. Then, $||v - v'|| = ||Tv - Tv'|| \leq \gamma||v - v'||$, or $(1-\gamma)||v - v'|| \leq 0$. Since a norm takes only non-negative values and $\gamma < 1$, we get that $||v - v'|| = 0$. Thus, $v - v' = 0$, or $v = v'$, finishing the proof of the first part of the statement.

For the second part, we have

$$\begin{aligned}
||v_n - v|| &= ||Tv_{n-1} - Tv|| \\
&\leq \gamma||v_{n-1} - v|| = \gamma||Tv_{n-2} - Tv|| \\
&\leq \gamma^2||v_{n-2} - v|| \\
&\vdots \\
&\leq \gamma^n||v_0 - v||. \quad \square
\end{aligned}$$

Lemma 1. *(Monotonicity of Bellman optimality operator)*

$$V(x) \leq V'(x) \implies T^*V(x) \leq T^*V'(x), \quad x \in \mathcal{X}$$

Proof:

$$\begin{aligned}
T^*V(x) &= \sup_{a\in\mathcal{A}} \big\{r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y)V(y)\big\} \\
&\leq \sup_{a\in\mathcal{A}} \big\{r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y)V'(y)\big\} \\
&= T^*V'(x) \quad \square
\end{aligned}$$

Lemma 2. *(Additivity of Bellman optimality operator) Let $d \in \mathbb{R}$ and $\mathbb{1} \in \mathbb{R}^{\mathcal{X}}$ be a vector of all ones, then*

$$T^*(V + d\mathbb{1})(x) = T^*V(x) + \gamma d\mathbb{1}.$$

Proof:

Note that the expectation of constant is itself. Therefore,

$$\begin{aligned}
T(V+d)(x) &= \sup_{a\in\mathcal{A}} \big\{r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y)(V(y)+d)\big\} \\
&= \sup_{a\in\mathcal{A}} \big\{r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y)V(y) + \gamma d\big\} \\
&= T^*V(x) + \gamma d \quad \square
\end{aligned}$$

Corollary 1. *Bellman optimality operator is contraction mapping in $||\cdot||_\infty$norm:*

$$||T^*V - T^*V'||_\infty \leq \gamma||V - V'||_\infty.$$

Proof:

Let $d = ||V(x) - V'(x)||_\infty$. Then,

$$V(x) - d \leq V'(x) \leq V(x) + d, \quad x \in \mathcal{X}$$

Apply T^* to both sides and use monotonicity and additivity to get

$$T^*V(x) - \gamma d \leq T^*V'(x) \leq T^*V(x) + \gamma d, \quad x \in \mathcal{X}.$$

Therefore,

$$||T^*V(x) - T^*V'(x)||_\infty \leq \gamma d, \quad x \in \mathcal{X}. \quad \square$$

As corollary 1 shows, the Bellman optimality operator T^* is a γ-contraction. Therefore, the dynamical system $V_{k+1} = T^*V_k$ converges to a unique fixed point, where $V = T^*V$ holds. Because we already know that V^* satisfies the Bellman optimality equation, V^* is indeed the unique solution to which the dynamical system converges after iterative updates. This proof concludes the description of value iteration.

1.5.2 Policy iteration

Now, we explain policy iteration. Policy iteration consists of two components: policy evaluation and policy improvement. First you start with a random policy and obtain the value function of that policy, then find a new policy that is better than the previous one in terms of the value function. Policy iteration repeats these two steps until the policy stops improving. Formally, after initializing the policy arbitrarily, we let

$$\begin{aligned}\pi_{k+1}(x) &= \arg\max_{a \in \mathcal{A}} Q^{\pi_k}(x, a) \\ &= \arg\max_{a \in \mathcal{A}} r(x, a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, a, y) V^{\pi_k}(x),\end{aligned}$$

where V^{π_k} is obtained by policy evaluation. In the rest of this section, We first elaborate on the policy evaluation and policy improvement steps, then show this algorithm surely converges to the optimal one.

Policy evaluation is similar to solving the dynamical system $V_{k+1} = T^*V_k$. Let us define another operator:

Definition 18. *(Bellman operator) Bellman operator underlying π, $T^\pi : \mathbb{R}^{\mathcal{X}} \to \mathbb{R}^{\mathcal{X}}$ is defined by*

$$T^\pi V(x) = r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V(y).$$

By the same argument as the Bellman optimality operator, the Bellman operator is also a γ contraction, so that the dynamical system converges to the fixed point $V = T^\pi V$, where $V = V^\pi$.

Having said that, note that the Bellman operator T^π is an affine operator, unlike T^*, so $V = T^\pi V$ can also be solved analytically. By stacking all states into a matrix or a vector, define $\mathbf{P} \in \mathbb{R}^{\mathcal{X} \times \mathcal{X}}$ where $\mathbf{P}[x, y] = \mathcal{P}(x, \pi(x), y)$, $\mathbf{v} \in \mathbb{R}^{\mathcal{X}}$ where $\mathbf{v}[x] = V(x)$, and $\mathbf{r} \in \mathbb{R}^{\mathcal{X}}$ where $\mathbf{r}[x] = r(x, \pi(x))$. Then, the Bellman equation becomes

$$\mathbf{v} = \mathbf{r} + \gamma \mathbf{P} \mathbf{v},$$

therefore,

$$\mathbf{v} = (\mathbf{I} - \gamma \mathbf{P})^{-1} \mathbf{r}.$$

The efficiency depends on the scale of the problem. When the state-action space is small, the analytic approach is straightforward, but when the space is large, the iterative approach becomes more efficient.

After the policy evaluation step, we can iteratively improve the current policy by policy improvement. A naive question for updating a policy is the following. At a state x, would taking an action a, which is different from the current policy $\pi(x)$, increase the expected return? In other words, is $Q^\pi(x,a)$ greater than $V^\pi(x)$? If so, intuitively, it would be better to have a new policy π' that selects a every time when x is encountered, i.e. $\pi'(x) = a$. This intuition is correct and can be proved as a special case of a general result called the policy improvement theorem. See [5] for the proof.

Proposition 1. *(Policy improvement) Let π and π' be any pair of deterministic policies such that,*

$$Q^\pi(x, \pi'(x)) \geq V^\pi(x), \quad \forall x.$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $x \in \mathcal{X}$:

$$V^{\pi'}(x) \geq V^\pi(x).$$

This result can be applied to updates in all states and all possible actions. We can consider selecting at each state the action that appears the best according to $Q^\pi(x,a)$, i.e. following the new greedy policy $\pi'(x) = \arg\max_a Q^\pi(x,a)$. According to the proposition 1, the greedy policy is no worse than the original policy.

Once a policy, π, has been improved using V^π to yield a better policy, π', we can then compute $V^{\pi'}$ and improve the policy again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies by repetition of policy evaluation and policy improvement. Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy in a finite number of iterations.

1.5.3 Value iteration vs. policy iteration

Finally, we present the pseudo code for value iteration and policy iteration and a brief analysis.

Algorithm 1 Value iteration

```
Require: $\mathcal{P}$ is the state transition probability kernel, $r$ is the immediate reward function
  1. Initialize array $V$ arbitrarily
  2. Update values for each state until convergence
  while $\Delta > \epsilon$ do
      $\Delta \leftarrow 0$
      for each $x \in \mathcal{X}$ do
          $U(x) \leftarrow V(x)$
          $V(x) \leftarrow \max_{a \in \mathcal{A}} \{ r(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,a,y) U(y) \}$
          $\Delta \leftarrow \max(\Delta, |U(x) - V(x)|)$
      end for
  end while
  3. Output a deterministic policy, $\pi$, such that
         $\pi(x) = \arg\max_{a \in \mathcal{A}} \{ r(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,a,y) V(y) \}$
```

Value iteration is obtained by applying the Bellman optimality operator to the value function in an iterative manner (algorithm 1, step 2), and it eventually converges to the optimal value. We can get the optimal policy out of it as the one that is greedy with respect to the optimal value function for every state (algorithm 1, step 3).

Algorithm 2 Policy iteration

```
Require: $\mathcal{P}$ is the state transition probability kernel, $r$ is the immediate reward function.
  1. Initialize array $V$ and $\pi$ arbitrarily
  while policy is not converged do
      2. Policy evaluation
      while $\Delta > \epsilon$ do
          $\Delta \leftarrow 0$
          for each $x \in \mathcal{X}$ do
              $U(x) \leftarrow V(x)$
              $V(x) \leftarrow r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) U(y)$
              $\Delta \leftarrow \max(\Delta, |U(x) - V(x)|)$
          end for
      end while
      3. Policy improvement
      for each $x \in \mathcal{X}$ do
          $\pi(x) \leftarrow \arg\max_{a \in \mathcal{A}} \left\{ r(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,a,y) V(y) \right\}$
      end for
  end while
  4. Output converged $\pi$
```

On the other hand, policy iteration performs nested iterative steps. In the inner iteration, it evaluates the value function underlying the current policy by applying the Bellman operator iteratively (algorithm 2, step 2), and in the outer iteration, it updates the policy by taking the greedy action for all states (algorithm 2, step 3).

Notice that when we perform policy evaluation for some policy, we do not necessarily have to wait for the convergence in the inner iteration. We could stop at some earlier point and move on to the policy improvement step with the approximated value function. This makes sense because policies generally converge much faster than value functions. Even if the value function is not converged yet, the policy induced by the early-stopped value function may well be converged already and can be used for policy improvement. The catch is that it then requires a greater number of outer iterations until convergence. The extreme case of policy iteration where you always stop the inner iteration after one step is exactly the value iteration. In this sense, value iteration is a special case of policy iteration.

Then, how do they compare in terms of the computational complexity? The computational complexity of value iteration and the outer iteration of policy iteration is $\mathcal{O}(|S|^2|A|)$ (take the max for each state, for each action, for each destination), and the complexity of the inner iteration of policy iteration is $\mathcal{O}(|S|^2)$ (take the max for each state, for an action given by policy, for each destination). As you can see, the value iteration and the outer loop of policy iteration is expensive when action space is large. Policy iteration reduces the number of required outer iterations by having the cheaper inner iteration be a more efficient algorithm. Puterman states that "in practice, value iteration should never be used. Implementation of modified policy iteration requires little additional programming effort yet attains superior convergence [10]." How many inner iterations you should perform is an important question as it is a trade-off between the number of inner iterations and the number of outer iterations. There is some work that investigates this issue by combining value iteration and policy iteration algorithms [11].

This concludes the description of MDPs and the exact algorithms for complete MDPs. Next, we examine algorithms for incomplete MDPs.

2 Reinforcement learning

2.1 Predicting value functions

In reinforcement learning, if a perfect model is not thrown, meaning, an MDP is not be completely specified, then we need to apply some estimation methods to predict the model before policy iteration. In this section, we consider the problem of estimating the value function underlying an MRP (see definition 7). An MRP arises when you fix a policy π for your MDP.

Value prediction problems arise in a number of ways: Estimating the probability of some future event, the expected time until some event occurs, or the (action-)value function underlying some policy in an MDP are all value prediction problems. Specific applications include estimating the failure probability of a large power grid [12] or estimating taxi-out times of flights at busy airports [13].

The value of a state is defined as the expectation of the random return when the process is started from the given state. The so-called *Monte-Carlo method* computes an average over multiple independent realizations started from each state to estimate this value for each state. However, this method can suffer from high variance of the returns, which means the quality of estimates could be poor. Also, when interacting with a system in a closed-loop fashion (i.e., when estimation happens while interacting with the system), it might be impossible to reset the state of the system to some particular state. In this case, the Monte-Carlo technique cannot be applied without introducing some additional bias. *Temporal difference (TD) learning* is a method that can be used to address these issues.

2.1.1 TD in finite state spaces

The unique feature of TD learning is that it uses *bootstrapping*: predictions are used as targets during the course of learning. In this section, we first introduce the most basic TD(0) algorithm and explain how bootstrapping works (see Algorithm 3). Next, we compare TD learning to Monte-Carlo methods (see Algorithm 4). Finally, we present the TD(λ) algorithm that unifies the two approaches (see Algorithm 5), where λ is a hyperparameter of the algorithms.

Here we consider only the case of finite MRPs, when computing the value-estimates of all the states is computationally intractable. This is known as the *tabular case* in the reinforcement learning literature. We will discuss the infinite case in section 2.1.2.

Fix some finite MRP $\mathcal{M}$. We wish to estimate the value function V underlying $\mathcal{M}$ given a realization $((X_t, R_{t+1}),\, t \geq 0)$ of $\mathcal{M}$. We only take states and rewards into account here to estimate the value function. Let $\hat{V}_t(x)$ denote the estimate of state x at time t. In the t^{th} step TD(0) performs the following calculations:

$$\delta_{t+1} = R_{t+1} + \gamma \hat{V}_t(X_{t+1}) - \hat{V}_t(X_t),$$

$$\hat{V}_{t+1}(x) = \hat{V}_t(x) + \alpha_t \delta_{t+1} \mathbb{1}_{X_t = x},$$

$$x \in \mathcal{X}.$$

Here the *step-size* sequence $(\alpha_t; t \geq 0)$ consists of (small) non-negative numbers chosen by the user. Algorithm 3 shows the pseudo-code of this algorithm. The function presented must be called after each transition.

Algorithm 3 Tabular TD(0)

```
Require: X is the last state, Y is the next state, R is the immediate reward associated with this transition,
  V is the array storing the current value estimates.
  1. $\delta \leftarrow R + \gamma \cdot V[Y] - V[X]$
  2. $V[X] \leftarrow V[X] + \alpha \cdot \delta$
  3. return $V$
```

A closer inspection of the update equation reveals that the only value changed is the one associated with X_t, i.e., the state just visited (cf. line 2 of the pseudo-code). Further, when $\alpha_t \leq 1$, the value of X_t is moved towards the target $R_{t+1} + \gamma \hat{V}_t(X_{t+1})$. Since the target depends on the estimated value function, the algorithm uses *bootstrapping*. The term temporal difference in the name of the algorithm comes from that δ_{t+1} is defined as the difference between values of states corresponding to successive time steps.

One can also estimate the value of a state by computing sample means, giving rise to the so-called *every visit Monte-Carlo method*. Let the underlying MRP be $\mathcal{M} = (\mathcal{X}, \mathcal{P}_0)$ and let $((X_t, R_{t+1}, Y_{t+1}); t \geq 0)$ be generated by continual sampling in $\mathcal{M}$ with restarts from some distribution P_0 defined over $\mathcal{X}$. Y represents the next state here. Let $(T_k; k \geq 0)$ be the sequence of times when an episode starts (thus, for each k, X_{T_k} is sampled from $\mathcal{P}_0$). For a given time t, let $k(t)$ be the unique episode index such that $t \in [T_k, T_{k+1})$. Let $G_t = \sum_{s=t}^{T_{k(t)+1}-1} \gamma^{s-t} R_{s+1}$ denote the return from time t on until the end of the episode. $V(x) = E[G_t | X_t = x]$ for any state x such that $P(X_t = x) \geq 0$. Hence, we could update the estimates as follows

$$\hat{V}_{t+1}(x) = \hat{V}_t(x) + \alpha_t (G_t - \hat{V}_t(x)) \mathbb{1}_{X_t = x}, x \in \mathcal{X}.$$

Monte-Carlo methods such as the above one are called *multi-step methods*, because they use multi-step predictions of the value. The pseudo-code of this update-rule is shown in Algorithm 4. The routine presented in the algorithm must be called at the end of each episode with the state-reward sequence collected during the episode. Note that the algorithm as shown here has linear time- and space-complexity in the length of the episodes.

Algorithm 4 Every-visit Monte-Carlo

```
Require: $X_t$ is the state at time t, $R_{t+1}$ is the reward associated with the $t^{th}$ transition, T is the length of
  the episode, V is the array storing the current value function estimate.
  $sum \leftarrow 0$
  for $t \leftarrow T-1$ downto 0 do
      $sum \leftarrow R_{t+1} + \gamma \cdot sum$
      $target[X_t] \leftarrow sum$
      $V[X_t] \leftarrow V[X_t] + \alpha \cdot (target[X_t] - V[X_t])$
  end for
```

Both Monte-Carlo and TD(0) are instances of stochastic approximation and achieve the same goal. Interestingly, there is a way to unify these approaches. This is achieved by the so-called $TD(\lambda)$ family of methods. Here, $\lambda \in [0, 1]$ is a parameter that allows one to interpolate between the Monte-Carlo and TD(0) updates: $\lambda = 0$ gives TD(0), while TD(1) is equivalent to a Monte-Carlo method. Actually, $TD(\lambda)$ can be viewed as the learning analogue to value iteration using dynamic programming (review section 1.5).

Given some $\lambda > 0$, the targets in the $TD(\lambda)$ update are given as some mixture of the multi-step return predictions $\mathcal{R}_{t;k} = \sum_{s=t}^{t+k} \gamma^{s-t} R_{s+1} + \gamma^{k+1} \hat{V}_t(X_{t+k+1})$, where the mixing coefficients are the exponential weights $(1-\lambda)\lambda^k, k \geq 0$. Thus, for $\lambda > 0$, $TD(\lambda)$ will be a multi-step method. The algorithm is made incremental by the introduction of the so-called *eligibility traces*.

In fact, the eligibility traces can be defined in multiple ways and hence $TD(\lambda)$ exists in correspondingly many forms. The update rule of $TD(\lambda)$ with the so-called *accumulating traces* is as follows:

$$\begin{gathered}
\delta_{t+1} = R_{t+1} + \gamma \hat{V}_t(X_{t+1}) - \hat{V}_t(X_t), \\
z_{t+1}(x) = \mathbb{1}_{x = X_t} + \gamma \lambda z_t(x), \\
\hat{V}_{t+1}(x) = \hat{V}_t(x) + \alpha_t \delta_{t+1} z_{t+1}(x), \\
z_0(x) = 0, \\
x \in \mathcal{X}.
\end{gathered}$$

Here $z_t(x)$ is the *eligibility trace* of state x. The rationale of the name is that the value of $z_t(x)$ modulates the influence of the TD error on the update of the value stored at state x. In another variant of the algorithm, the eligibility traces are updated according to

$$z_{t+1}(x) = max(\mathbb{1}_{x = X_t}, \gamma \lambda z_t(x)), \quad x \in \mathcal{X}.$$

This is called *replacing traces* update. Algorithm 5 gives the pseudo-code corresponding to the variant with replacing traces. The function presented must be called after each transition. In practice, the best value of λ is determined by trial and error.

Algorithm 5 Tabular $TD(\lambda)$ with Replacing Traces

```
Require: X is the last state, Y is the next state, R is the immediate reward associated with this transition,
  V is the array storing the current value function estimate, z is the array storing the eligibility traces.
  $\delta \leftarrow R + \gamma \cdot V[Y] - V[X]$
  for all  $x \in \mathcal{X}$ do
      $z[x] \leftarrow \gamma \cdot \lambda \cdot z[x]$
      if X = x then
          $z[x] \leftarrow 1$
      end if
      $V[X] \leftarrow V[X] + \alpha \cdot \delta \cdot z[x]$
  end for
  return $(V, z)$
```

In summary, $TD(\lambda)$ allows one to estimate value functions in MRPs. It generalizes Monte-Carlo methods, can be used in non-episodic problems, and allows for bootstrapping. Further, by appropriately tuning λ it can converge significantly faster than Monte-Carlo methods or TD(0).

2.1.2 $TD(\lambda)$ with function approximations

When the state space is large (or infinite), it is not feasible to keep a separate value for each state in the memory. Clearly, in this case, one must compress the table representing the values. Abstractly, this can be done by relying on an appropriate function approximation method. In such cases, we often seek an estimate of the values in the form

$$V_\theta(x) = \theta^T \phi(x), x \in \mathcal{X},$$

where $\theta \in \mathbb{R}^{\mathrm{d}}$ is a vector of parameters and $\phi : \mathcal{X} \to \mathbb{R}^{\mathrm{d}}$ is a mapping of states to d-dimensional vectors. For state x, the components of the vector $\phi(x)$ are called the *features* of state x and ϕ is called a *feature extraction* method. The individual functions ϕ_i are called *basis functions.*

Let us return to the problem of estimating a value function V of a Markov reward process $\mathcal{M} = (\mathcal{X}, \mathcal{P}_0)$, but now assume that the state space is large (or even infinite). Let $\mathcal{D} = ((X_t, R_{t+1}); t \geq 0)$ be a realization of $\mathcal{M}$. The goal is to estimate the value functions of $\mathcal{M}$ given $\mathcal{D}$ in an incremental manner. Choose a smooth parametric function approximation method $(V_\theta; \theta \in \mathbb{R}^d)$. For example, for any $\theta \in \mathbb{R}^d$, $V_\theta : \mathcal{X} \to \mathbb{R}$ is such that $\nabla_\theta V_\theta(x)$ exists for any $x \in \mathcal{X}$. The generalization of tabular $TD(\lambda)$ with accumulating eligibility traces to the case when the value functions are approximated using members of $(V_\theta; \theta \in \mathbb{R}^d)$ uses the following updates:

$$\delta_{t+1} = R_{t+1} + \gamma V_{\theta_t}(X_{t+1}) - V_{\theta_t}(X_t),$$

$$z_{t+1} = \nabla_\theta V_{\theta_t}(X_t) + \gamma \lambda z_t,$$

$$\theta_{t+1} = \theta_t + \alpha_t \delta_{t+1} z_{t+1},$$

$$z_0 = 0.$$

Here $z_t \in \mathbb{R}^d$. Algorithm 6 shows the pseudo-code of this algorithm. The function presented must be called after each transition.

Algorithm 6 $TD(\lambda)$ with Linear Function Approximation

```
Require: X is the last state, Y is the next state, R is the immediate reward associated with this transition,
    $\theta$ is the parameter vector of the linear function approximation, z is the vector of eligibility traces.
  1. $\delta \leftarrow R + \gamma \cdot \theta^T \phi[Y] - \theta^T \phi[X]$
  2. $z \leftarrow \phi[X] + \gamma \cdot \lambda \cdot z$
  3. $\theta \leftarrow \theta + \alpha \cdot \delta \cdot z$
  4. return $(\theta, z)$
```

To see that this algorithm is indeed a generalization of tabular $TD(\lambda)$ assuming that $\mathcal{X} = \{x_1, \cdots, x_D\}$ and let $V_\theta(x) = \theta^T \phi(x)$ with $\phi_i(x) = \mathbb{1}_{x=x_i}$. Note that V_θ is linear in the parameters, it holds that $\nabla_\theta V_\theta = \phi$. Hence, we see that the update above indeed reduces to the previous one.

In the off-policy version of $TD(\lambda)$, the definition of δ_{t+1} becomes:

$$\delta_{t+1} = R_{t+1} + \gamma V_{\theta_t}(Y_{t+1}) - V_{\theta_t}(X_t).$$

Unlike the tabular case, under off-policy sampling, convergence is no longer guaranteed, and the parameters may diverge. This is true for linear function approximation when the distributions of $(X_t; t \geq 0)$ do not match the stationary distribution of the MRP $\mathcal{M}$. Another case when the algorithm may diverge is when it is used with a nonlinear function approximation method.

On the positive side, almost sure convergence can be guaranteed when (i) a linear function approximation method is used with $\phi : \mathcal{X} \rightarrow \mathbb{R}^d$; (ii) the stochastic process $(X_t; t \geq 0)$ is an ergodic Markov process whose stationary distribution μ is the same as the stationary distribution of the MRP $\mathcal{M}$; and (iii) the step-size sequence satisfies the Robbins-Monro conditions (square summable, but not summable). In the results cited, it is also assumed that the components of ϕ are linearly independent. When this holds, the limit of the parameter vector will be unique. In the other case, i.e., when the features are redundant, the parameters will still converge, but the limit will depend on the parameter vectors initial value. However, the limiting value function will be unique.

The solution obtained by TD(0) can be thought of as the solution of a deterministic MRP with a linear dynamics. In fact, this also holds in the case of $TD(\lambda)$. This suggests that if the deterministic MRP captures the essential features of the original MRP, V_{θ^λ} will be a good approximation to V. As λ gets closer to 1, it becomes more important for the features to capture the structure of the value function, and as λ gets closer to 0, it becomes more important to capture the structure of the immediate rewards and the immediate feature-expectations. This suggests that the best value of λ (i.e., the one that minimizes *Bellman error* $||\Delta^{(\lambda)}(V_{\theta^{(\lambda)}})||$) may depend on whether the features are more successful at capturing the short-term or the long-term dynamics (and rewards).

2.2 Model-based methods

Let us now move from estimating value functions in MRPs to finding policies in MDPs. The space of learning problems can be split based on the interactions that the agent has with the system into two classes of problems:

- Interactive problems: Where the learner can influence the observation
- Non-Interactive problems: Where learner cannot manipulate the observations and the data is prepared before learning.

These problems use the reward and policy model that is calculated for MDP to find the optimal policy. The goals of these problems can differ slightly from one another. The goal of non-interactive problems is mostly to find the best policy, often with time and memory constraints. For instance, we might want to learn a sufficiently good policy as quickly as possible.

Interactive learning problems can be further split up based on their goals. The first class of problems are online learning problems, whose goal is to increase online performance, such as number of optimal solutions or reward gain.

Another category of interactive problems is active learning. Its goal is to find the best policy for the problem, similar to the non-interactive problems. The main difference between the two category is that in active learning problems, the agent can influence the input in favor of finding a better policy. Specifically, based on what examples have been seen so far, the agent can adjust its sampling strategy more towards either exploration exploitation.

In the following section we will consider problems of online learning and active learning, for the both scenario of having a bandit(i.e. an MDP with single state) and normal MDPs.

2.2.1 Online learning in Bandits

In this section we first look closely at the problem of online learning in bandits. Bandits are MDPs with only one state and multiple actions. Once we have learned an optimal value function, we want to follow a greedy approach to guarantee the best results. However, during training, before the value function has converged, taking a greedy approach can cause the learner to get stuck in a local optimum. Thus each learner should be able to *explore* by trying sub-optimal actions. However, we also don't want the learner to spend all its time taking sub-optimal actions, so we want to limit how many explore actions the learner takes. There are a variety of methods for choosing how to balance exploration and exploitation.

ϵ-Greedy Method In this method we choose to explore with probability of some $\epsilon \in [0, 1]$. Specifically, with probability ϵ take a random action, and with probability $1 - \epsilon$ take the optimal action. Typically, ϵ decays over time, allowing the learner to eventually hone in on the optimal policy.

Boltzmann exploration The main difference between this strategy and the ϵ-Greedy one is that in this method we can take the values of each action into account. With this strategy the action at each stage is chosen according to the following distribution:

$$\pi(a) = \frac{exp(\beta Q_t(a))}{\sum_{a' \in A} exp(\beta Q_t(a'))}$$

In the equation above, $Q_t(a)$ is the sample mean of action a up to time t, and β is a parameter that determines the exploration-exploitation trade-off. The intuition behind this formula is to weight actions according to their relative values. Higher values of β result in a greedier policy, with $\beta \to \infty$ resulting in the greedy policy of always selecting the optimal action.

Optimism in the face of uncertainty While the previous two approaches can be competitive with more complicated methods, there is no known automated way of properly adjusting the parameters ϵ or β to achieve optimal performance. This approach is potentially a better implementation of the exploration-exploitation strategy. According to this strategy the learner should choose an action that has the highest value for the upper confidence bound. The main intuition is to check the history of each action and choose them based on how well they get rewarded over a period of time with consideration of the current reward. There are different method for calculating the upper bound but the best one is UCB1. According to that the upper bound can be computed as below.

$$U_t(a) = r_t(a) + R\sqrt{\frac{2 \log t}{n_t(a)}}$$

In the equation above $n_t(a)$ is the number of times that action a is chosen until time t, $r_t(a)$ shows the average reward of taking action a until time t, and R is an upper bound for the observed rewards. If we know that the variance of reward values are not high we can replace R with estimates of the variance as it will be a better upper bound. Using this UCB as a sampling strategy is done in two stages: *select* and *update*. In the select phase, we choose the action that has the highest upper bound. During the update stage we update the count and average reward based on the selected action.

2.2.2 Online learning in MDPs

Now lets consider the problem of online learning for MDPs. The goal that we are pursuing here is to minimize the regret. Regret is the difference between the total reward that our learner receives and the optimal reward that the learner could have received. Let's consider the MDP that has all the states connected (unichain) and the immediate rewards are all in the interval of [0,1]. For a fixed policy, π on this MDP, we can define a stationary distribution over the states which we will call $\mu_\pi(x)$. Therefore the average reward for policy π would be:

$$\rho^\pi = \sum_{x \in X} (\mu_\pi(x))(r(x, \pi(x)))$$

Here r is the reward function for taking action $\pi(x)$ at state x. If we call the optimal long-time reward $\rho^* = \arg\max_\pi \rho^\pi$ then we can define the regret to be:

$$\mathbf{R}_T^A = T\rho^* - \mathcal{R}_T^A$$

In equation above $\mathbf{R}_T^A$ is regret and $\mathcal{R}_T^A$ is the reward of using some learning algorithm A until time T. As can be seen above, minimizing the regret is exactly maximizing the total reward obtained over time.

UCRL2 There exists an algorithm called UCRL2 [14] that achieves logarithmic regret with respect to time. In order to describe this algorithm in detail we have to go over some definitions:

Definition 19. *(Diameter) The diameter is the largest number of steps, in expectation under a policy minimizes the number of steps, that is needed to get to some state starting from some other state. In other words this is a longest path in the markov chain. Note that if an MDP has any states that are not reachable from some other state, the diameter is infinite.*

Definition 20. *(Gap) The gap is the difference between the optimal policy and the second optimal performance.*

By setting the confidence parameter to $\lambda = 1/(3T)$, the expected regret of the UCLR2 algorithm is:

$$E[\mathbf{R}_T^{UCRL2(\lambda)}] = O(\frac{D^2|X|^2|A|\log(T)}{g})$$

Where D is the diameter of the MDP, and g is the gap. This expected regret can be used in the situations where we have a high value for g. For small g since the value of the expected regret gets large there would be no point for using the bound with small T. In those cases there is an alternative expected regret defined as:

$$E[\mathbf{R}_T^{UCRL2(\lambda)}] = O(D|X|\sqrt{|A|T\log(T)})$$

UCRL2 algorithm has two steps. In the first step each action is executed and the next state and reward are calculated. This algorithm constructs confidence intervals around the estimates of the transition probabilities and immediate reward function. These define a set of plausible MDPs. One upside of this algorithm is that it waits to update the policy until sufficient information has been obtained, as can be seen in line 6 of the pseudo-code below. Here, n_2 is a counter for (state, action) pairs, n_3 is a counter for (state, action, next state) triples. n_2' and n_3' count the number of times each pair/triple has been seen since the last update. r keeps track of the rewards obtained for starting with each (state, action) pair.

Algorithm 7 UCLR2 algorithm.

```
Require: $\lambda \in [0,1]$ is a confidence parameter
 1: for all $x \in X : \pi[x] \leftarrow a_1$
 2: $n_2, n_3, r, n'_2, n'_3 \leftarrow 0$
 3: $t \leftarrow 1$
 4: Repeat
 5:     $A \leftarrow \pi[X]$
 6:     if $n'_2[X,A] > max(1, n_2[X,A])$
 7:         $n_2 = n'_2 + n_2, n_3 = n'_3 + n_3, r = r + r'$
 8:         $n'_2, n'_3, r = 0$
 9:         $\pi = OptSolve(n_2, n_3, r, \lambda, t, )$
10:         $A \leftarrow \pi[X]$
11:     $(R, Y) \leftarrow ExecuteInWorld(A)$
12:     $n'_2[X,A] \leftarrow n'_2[X,A] + 1$
13:     $n'_3[X,Y,A] \leftarrow n'_3[X,Y,A] + 1$
14:     $r[X,A] \leftarrow r[X,A] + R$
15:     $X \leftarrow Y$
16:     $t = t + 1$
```

OptSolve works by computing a near-optimal policy (how near depends on the confidence parameter λ) and an *optimistic model* by running value iteration over a special MDP. This special MDP is defined such that the state space is the same as the actual MDP, but the actions become continuous pairs (A, p), where p is a distribution over next states. This new action space is defined based on the confidence bounds over the real transition probabilities computed from the data seen so far. Once the policy is computed, it is executed over the actual MDP to acquire new examples with which to update the confidence bounds.

2.2.3 Active learning in Bandits

Let's consider the active learning strategy in bandits. The goal is to maximize the reward over T interactions. If we have the upper and lower bounds for each action at each time we can find the actions that can be eliminated. If the upper bound of an action at time t is less than the max of the lower bounds of other actions, i.e. $U_t(a) < Max_{a' \in A} L_t(a^{'})$ The action can be eliminated with certainty. Here is how we can calculate the upper and lower bounds: In these formulas λ shows the certainty metric for the action.

$$U_t(a) = Q_t + R\sqrt{\frac{\log(2|A|T/\lambda)}{2n_t(a)}}$$

$$L_t(a) = Q_t - R\sqrt{\frac{\log(2|A|T/\lambda)}{2n_t(a)}}$$

2.2.4 Active learning in MDPs

Let's consider a deterministic MDP with n states and m actions. We want to learn a near optimal policy in as few time steps as possible. We can first recover the MDP's transition structure using the following process: for each state-action that is not considered we should get to that state and execute that action. Getting from any state to any other state will take at most $n-1$ steps. Since there are n states and m actions the algorithm will take $O(n^2m)$ to finish. Given the transition structure, we can obtain estimates of the reward structure by simply taking each action repeatedly, similar to the bandit case. This yields an estimate of each state-action reward of accuracy ϵ with probability $1-\delta$ after at most $\log(nm/\delta)$ visits to each state-action pair. Having this model then allows us to compute a near optimal policy over the MDP.

Specifically, we can compute an ϵ-optimal policy in at most $n^2m + 4e\log(nm/\delta)/((1-\gamma)^2\epsilon)^2$ steps, where $e \leq n^2m$ is the number of time steps to visit all state-action pairs.

There is currently no similar result for active learning in stochastic MDPs.

2.3 Model-free methods

Let us now discuss a set of methods that approximate the action-value function, $Q(x,a)$, directly, rather than estimating the state-value function, $V(x)$. Because these methods directly learn the action-value function without estimating a model of the MDP's transition or reward structures, they are known as "Model-free" approaches.

Assuming we are able to learn a near-optimal Q function, we can give a lower bound on value of a policy, π that is greedy w.r.t. Q. As shown in [15], the value of policy π can be bounded as follows:

$$V^{\pi}(x) \geq V^*(x) - \frac{2}{1-\gamma}\left|\left|Q - Q^*\right|\right|_{\infty}, x \in \mathcal{X}$$

Which shows that as the learned Q function approaches Q^*, the value, V^{π} of following a greedy policy with respect to the laerned Q function approaches V^*.

2.3.1 Q-learning

The first model-free approach for learning policies is Q-learning. Similar to the methods described above, Q-learning can be applied in both finite and infinite MDPs. We first look at the finite setting, and then generalize that with various function approximations to apply the approach to the infinite setting.

Finite MDPs Q-learning was first introduced in [16], and uses the TD approach to estimate the Q function. Upon observing a transition $(X_t, A_t, R_{t+1}, Y_{t+1})$, the estimates are updated as follows:

$$\delta_{t+1}(Q) = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(Y_{t+1}, a') - Q(X_t, A_t),$$
$$Q_{t+1}(x,a) = Q_t(x,a) + \alpha_t \delta_{t+1}(Q_t)\mathbb{1}_{\{x=X_t, a=A_t\}}$$

It is proven in [?] that the above updates will converge if appropriate learning rates, α_t are used. It is simple to see that when it converges, it converges to the optimal Q^*. At convergence, it is necessary that $\mathbb{E}[\delta_{t+1}(Q)|X_t, A_t] = 0$. Using the fact that $(Y_{t+1}, R_{t+1}) \sim \mathcal{P}_0(\cdot|X_t, A_t)$, we can break this expectation down as follows

$$\begin{aligned}
\mathbb{E}[\delta_{t+1}(Q)|X_t, A_t] &= \mathbb{E}[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(Y_{t+1}, a') - Q_t(X_t, A_t)] \\
&= r(X_t, A_t) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(X_t, A_t, y) \sup_{a' \in \mathcal{A}} Q_t(y, a') - Q_t(X_t, A_t) \\
&= T^* Q_t(X_t, A_t) - Q_t(X_t, A_t)
\end{aligned}$$

Thus, when these updates converge, we get that $T^*Q_t(x,a) = Q_t(x,a)$, which is satisfied by $Q_t = Q^*$. Additionally, using the above equation, we can see that

$$\mathbb{E}[Q_{t+1}(X_t, A_t)|X_t, A_t] = (1-\alpha_t)Q_t(X_t, A_t) + \alpha_t T^* Q_t(X_t, A_t)$$

which shows that the expected update at each timestep is simply taking steps according to the Bellman optimality operator given the observed state-action pair.

Some of the benefits of Q-learning include its simplicity, as well as the fact that it is able to make use of arbitrary training data, as long as in the limit all state-action pairs are updated infinitely often. Similar to the exploration-exploitation trade-off seen in section 2.2.1, ϵ-greedy or Boltzmann approaches are often used to sample actions in the closed-loop setting.

Q-learning with function approximation Similar to the state-value learning approaches above, Q-learning can also be performed when the state or action spaces are very large, or even infinite, by using a function approximation of the Q function. The general approach to this is to replace Q with some parameterized Q_θ. The update for θ is similar to above:

$$\theta_{t+1} = \theta_t + \alpha_t \delta_{t+1} \nabla Q_{\theta_t}(X_t, A_t)$$

For example, using a linear approximation, $Q_\theta = \theta^{\mathrm{T}}\phi(X, A)$, where $\theta \in \mathcal{R}^d$ and $\phi : \mathcal{X} \times \mathcal{A} \to \mathcal{R}^d$, would result in the following update:

$$\begin{aligned}\delta &= R + \gamma \cdot \max_{a' \in \mathcal{A}} \theta^{\mathrm{T}}\phi(Y, a') - \theta^{\mathrm{T}}\phi(X, A) \\ \theta &= \theta + \alpha \cdot \delta \cdot \phi(X, A)\end{aligned}$$

Although this is widely used in practice, not much can be said about its convergence properties. In order to guarantee optimal convergence, it is necessary to either restrict the value function approximation, or modify the update procedure.

One such approach to restricting the value function is to treat Q_θ as a state and action aggregator, meaning the function ϕ partitions the state-action space, and returns a 1-hot vector indicating the region in which the state-action pair falls. Under this simplifying restriction, if the training sequence $((X_t, A_t); t \geq 0)$ is stationary, meaning the policy is fixed, the algorithm will behave exactly like Q-learning in an induced MDP over the partitions, and will thus converge to an approximation of the actual Q^*. A further improvement on this is to treat the partitions as a spectrum, and return a vector such that $\sum_{i=1}^{d} \phi_i(x, a) = 1$. This allows the approximation to avoid issues with hard boundaries between partitions. The update rule is then modified to only update one component of θ_t by randomly selecting an index from the distribution $(\phi_1(X_t, A_t), ..., \phi_d(X_t, A_t))$.

2.3.2 Actor-critic methods

Another model-free way of directly learning policies is a set of approaches known as actor-critic methods. Actor-critic methods are a generalization of policy iteration. Basic policy iteration has two steps; a complete policy evaluation step, and a complete policy update step. However, exact evaluation of policies is not always possible, such as when using sample-based methods or function approximation. Actor-critic methods address this problem by updating the policy before it is fully evaluated, which is also known as generalized policy iteration. Note that due to updating partially evaluated policies, the generated policies do not monotonically improve over time. To address this, it is common to keep track of a sequence of policies during training, and later selecting on that performs the best empirically.

Actor-critic methods are able to perform off-policy learning, meaning that although the method is attempting to learn an optimal target policy for the MDP, there can be a separate policy used to generate training examples called the behavior policy. This allows the model to trade off between exploration and exploitation during training, and the behavior policy commonly uses techniques such as ϵ-greedy policies or Boltzmann exploration.

As the name implies, the basic setup is to have two separate algorithms, the actor and the critic. The purpose of the actor is to update the target policy, while the critic evaluates that policy.

Example critic method: SARSA Critic methods are just generalizations of the state-value estimation methods described in previous sections that estimate $Q(x, a)$ functions instead of $V(x)$. Note that all such methods for estimating V can be viewed as special cases of those estimating Q in which there is only one possible action at each state.

One such method is SARSA, named such due to making use of the current **S**tate, current **A**ction, next **R**eward, next **S**tate, and next **A**ction. Basic SARSA is just TD(0) applied to state-action pairs, and can be used to evaluate a specific policy. It follows the update below.

$$\delta_{t+1}(Q) = R_{t+1} + Q(Y_{t+1}, A_{t+1}) - Q(X_t, A_t)$$

$$Q_{t+1}(x,a) = Q_t(x,a) + \alpha_t \delta_{t+1}(Q_t) \mathbb{1}_{\{x=X_t, a=A_t\}}$$

This is very similar to Q-learning. The primary difference is that SARSA updates the Q function based on the next action taken according to the behavior policy, while Q-learning uses the greedy action. This potentially results in slightly different convergence values. For example, using an ϵ-greedy behavior policy with fixed ϵ, the expected value under Q-learning will be the optimal action value, $Q(s_t, a_t) = \max_a Q(s_t, a)$, while the expected value under SARSA is the weighted sum of the average action value and the optimal action value, $Q(s_t, a_t) = \epsilon \sum_a Q(s_t, a) + (1-\epsilon) \max_a Q(s_t, a)$.

Additionally, as SARSA is just TD(0) for state-action pairs, it can be also be extended to SARSA(λ), and use function approximation. These extensions are subject to the same limitations as in the TD(λ) case.

Example actor method: Greedy policy improvement The actor portion of actor-critic methods is responsible for improving a policy given the results of the critic's evaluation. The simplest such method is to simply take a greedy approach w.r.t. the action-value function returned by the critic. Note that with finite action space, this policy can be computed as needed, allowing it to run in very large state spaces.

References

[1] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

[2] Ronald A Howard. *Dynamic programming and Markov processes.* Wiley for The Massachusetts Institute of Technology, 1964.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[4] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[5] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[6] Eugene A Feinberg. Total expected discounted reward mdps: existence of optimal policies. *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[7] Richard Bellman. *Dynamic programming.* Courier Corporation, 2013.

[8] Stuart Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002.

[9] Emo Todorov. Markov decision processes and bellman equations. 2014.

[10] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.

[11] Elena Pashenkova, Irina Rish, and Rina Dechter. Value iteration and policy iteration algorithms for markov decision problem. In *AAAI96: Workshop on Structural Issues in Planning and Temporal Reasoning.* Citeseer, 1996.

[12] Mannor S. Frank, J. and D. Precup. Reinforcement learning in the presence of rare events. *Proceedings of the 25th In- ternational Conference Machine Learning (ICML 2008)*, 307.

[13] Ganesan R. Sherry L. Balakrishna, P. and B. Levy. Estimating taxi-out times with a reinforcement learning algorithm. *27th IEEE/AIAA Digital Avionics Systems Conference*, pages 3.D.3–1–3.D.3–12, 2008.

[14] Peter Auer, Thomas Jaksch, and Ronald Ortner. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*, 11:1563–1600, 2010.

[15] Satinder P Singh and Richard C Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16(3):227–233, 1994.

[16] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK, 1989.