Draft: Review of Reinforcement Learning

Pavel Piliptchak

July 2019

1 Introduction

Reinforcement learning has seen a renewed wave of interest in recent years thanks to its incorporation of deep neural networks. This has resulted in a number of high-profile demonstrations of reinforcement learning algorithms reaching superhuman performance in games such as Atari [3], Go [5], DotA [4] , and StarCraft [7]. Meanwhile, there has also been a growing body of research that applies reinforcement learning to other domains, notably robotics [2, 1]. This work, while showing promise, still faces a number of fundamental challenges that are exacerbated by the robotics domain in particular. The goal of this document is to introduce these challenges and survey current research that seeks to address them.

2 Fundamentals

This section mainly serves as a partial summary of *Reinforcement Learning: An Introduction* by Sutton and Barto [6]. Its goal is to explain the basic mathematical structure of reinforcement learning problems and algorithms as well as set the stage for discussing current reinforcement learning research problems.

2.1 Markov Decision Processes

Let's begin by formulating the reinforcement learning problem. The objective of reinforcement learning is to train an *agent* to act on its *environment* in some desired way over time. An agent can be thought of as an algorithm that outputs values over time, while the environment changes as a result of these values (this is analogous to the controller-plant relationship in control theory). Formally, this agent-environment interaction is described by a *Markov decision process* (MDP), which is characterized by the 5 terms: $\mathcal{S}$, $\mathcal{A}$, $\mathcal{T}$, R, and γ. We'll describe each term below.

In an MDP, the agent is characterized by the set of all actions it can take, $\mathcal{A}$. The environment is characterized by the set of possible configurations (states) it can take, $\mathcal{S}$. At any discrete moment in time, t, the environment is in some state $s_t \in \mathcal{S}$, and the agent must choose some action $a_t \in \mathcal{A}$ to perform. At

the next time step, $t+1$, the environment transitions to some new state, s_{t+1}; and the agent must choose a new action to take, a_{t+1}. This process can repeat indefinitely, or until some terminal state, s_T, is reached.

$\mathcal{T}$ is the set of *transition probabilities* between states. These describe the dynamics of the MDP. Interactions are often stochastic in the real world—to model this, we treat future states as random variables that are realized whenever a transition occurs. So, at any time t we consider three random variables: S_t, A_t and S_{t+1} . We define a transition probability to be $P\{S_{t+1} = s'|S_t = s, A_t = a\}$ for some $s \in \mathcal{S}$, $a \in \mathcal{A}$, and $s' \in \mathcal{S}$. $\mathcal{T}$ maps every s, a, and s' to a transition probability. Note that we only condition S_{t+1} on the previous state and action, meaning the chain of transitions satisfies the *Markov property*. Also note that these transition probabilities could be different for every t. In this case, the MDP has *non-stationary* dynamics in the probabilistic sense.

Next, we look at R—the *reward function*. The reward function explicitly specifies the desired behavior of an agent. It maps each transition in $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ to some value $r \in \mathbb{R}$.[1] Since every transition has a corresponding reward, we can also compute the reward for a sequence of transitions. We call this sequence a *trajectory* or *rollout*. If $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$ denotes a particular trajectory, then its cumulative reward is:

$$R_\tau = \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t, s_{t+1}) \tag{1}$$

Here, $\gamma \in (0, 1]$ is the *discount factor*, which decays the reward values over time and gaurantees that R_τ is bounded in the case that $T = \infty$. We can choose γ to be 1 only if T is finite (i.e., if there is some goal state that ends the trajectory of the agent). The agent's goal becomes choosing actions that lead to trajectories that have high reward values.

We describe the agent's choices using a *policy*, $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$. For any t, the policy defines $P\{A_t = a|S_t = s\}$. Since π specifies the conditional distributions of A_t's, and $\mathcal{T}$ specifies the conditional distributions of S_t's, we have a complete probabilistic description of the MDP. For example, we could sample a particular trajectory $\tau = (s_0 \sim S_0, a_0 \sim A_0, s_1 \sim S_1, a_1 \sim A_1, \ldots)$. We can also find the probability of that trajectory occurring by finding the joint probability $P(S_0 = s_0, A_0 = a_0, S_1 = s_1, A_1 = a_1, \ldots)$. Taking advantage of the Markov property, we can write this more succinctly as:

$$p(\tau) = p(s_0) \prod_{t=0}^{T-1} \mathcal{T}(s_{t+1}, a_t, s_t)\pi(a_t, s_t) \tag{2}$$

Here, the initialization $p(s_0) = P\{S_0 = s_0\}$ can be defined arbitrarily. Having

[1] Sometimes, the reward is defined to be a random variable in order to handle probabilistic outcomes. We can handle this for now by defining our state-space so that each reward outcome has its own state

defined $p(\tau)$ and R_τ, we can determine the expected reward of our policy as:

$$\mathbb{E}[R_\pi] = \sum_{\tau \in \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \ldots} p(\tau) R_\tau \tag{3}$$

Finding a policy that maximizes this expectation will be the goal of this section.

2.2 Dynamic Programming

First we'll approach the problem with *dynamic programming*, which provides a basis for reinforcement learning. As a precursor, we will want to make calculating $\mathbb{E}[R_\pi]$ more convenient, since summing over all possible trajectories isn't feasible. We do this by reformulating the concept of expected reward as a function of either just state or state-action pairs. These are known as the *value function* and *action-value function* (or Q-function) respectively:

$$V_\pi(s) = \mathbb{E}[R_\pi | S_0 = s] \tag{4}$$

$$Q_\pi(s, a) = \mathbb{E}[R_\pi | S_0 = s, A_0 = a] \tag{5}$$

By conditioning the expectation on either a state or state-action pair, we've reduced the sum to only be over trajectories starting with a particular (s) or (s, a). On its own, this isn't much of an improvement. However, we can get something much more useful by combining these ideas with the following recursive definitions of R_τ and $p(\tau)$:

$$\begin{aligned} R_\tau &= \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t, s_{t+1}) = R(s_0, a_0, s_1) + \gamma \sum_{t=0}^{T'-1} \gamma^t R(s'_t, a'_t, s'_{t+1}) \\ &= R(s_0, a_0, s_1) + \gamma R_{\tau_1} \end{aligned}$$

$$\begin{aligned} p(\tau) &= p(s_0) \prod_{t=0}^{T-1} \mathcal{T}(s_{t+1}, a_t, s_t) \pi(a_t, s_t) \\ &= \mathcal{T}(s_1, a_0, s_0) \pi(a_0, s_0) p(s_0) \prod_{t=0}^{T'-1} \mathcal{T}(s'_{t+1}, a'_t, s'_t) \pi(a'_t, s'_t) \\ &= p(s_0, a_0, s_1) p(\tau_1) \end{aligned}$$

where τ_1 is defined to be the trajectory $(s'_0, a'_0, \ldots, s'_{T'}) = (s_1, a_1, \ldots, s_T)$. Substituting these definitions into V_π, we have:

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}[R_\pi | S_0 = s] \\
&= \sum_{\tau : s_0 = s} p(\tau) R_\tau \\
&= \sum_{a_0 \in \mathcal{A}, s_1 \in \mathcal{S}} \left(p(s, a_0, s_1) \sum_{\tau_1} p(\tau_1) \left(R(s, a_0, s_1) + \gamma R_{\tau_1} \right) \right) \\
&= \sum_{a_0 \in \mathcal{A}, s_1 \in \mathcal{S}} \left(p(s, a_0, s_1) \left(R(s, a_0, s_1) + \gamma \sum_{\tau_1} p(\tau_1) R_{\tau_1} \right) \right) \\
&= \mathbb{E}\left[R(s, A_0, S_1) + \gamma \mathbb{E}[R_\pi | S'_0 = S_1] \right] \\
&= \mathbb{E}[R(s, A_0, S_1) + \gamma V_\pi(S_1)] \qquad (6)
\end{aligned}$$

with the expectation over A_0 and S_1. This is the *Bellman equation* for V_π. Following a similar derivation, we can also find a Bellman equation for Q_π:

$$Q_\pi(s, a) = \mathbb{E}\left[R(s, a, S_1) + \gamma \mathbb{E}\left[Q_\pi(S_1, A_1) \right] \right] \qquad (7)$$

where the first expectation is over S_1, and the second is over A_1. Using these recursive definitions of V_π and Q_π, we can use a technique called *iterative policy evaluation* to compute V_π and Q_π in place of trying to compute $\mathbb{E}[R_\pi]$ directly. We'll demonstrate the approach using V_π. We start by noting that Equation (6) is analogous to a linear system of equations, where we would like to solve for V_π. We could use an iterative method for solving this linear system, and Equation (6) happens to also be a good update rule for such a method:

$$V_{k+1}(s) = \mathbb{E}[R(s, A_0, S_1) + \gamma V_k(S_1)] \qquad \forall s \in \mathcal{S} \qquad (8)$$

The resulting sequence of V_k's converges to the true solution of the linear system, V_π. This type of update rule is called a *Bellman backup* or *bootstrapping*. From here, the expected reward of the policy is computed by just taking an expectation over the initial state:

$$\mathbb{E}[R_\pi] = \mathbb{E}[V_\pi(S_0)]$$

The second component of dynamic programming is *policy iteration*. This is a technique for finding an optimal policy from a sub-optimal policy's value functions. First, note that the optimal policy π^* satisfies $\mathbb{E}[R_{\pi^*}] \geq \mathbb{E}[R_\pi]$, where π is any policy. For this to hold under any initialization S_0, we would need $V_{\pi^*}(s) \geq V_\pi(s)$ to hold for all s. Policy iteration starts with an arbitrary π_k and generates a new policy π_{k+1} such that $V_{\pi_{k+1}}(s) \geq V_{\pi_k}(s)$ for all s. Once $V_{\pi_k} = V_{\pi_{k+1}}$, the generated policy is equivalent to π^*.

How do we update π_k in a way that consistently improves V_{π_k}? We can do this by looking for discrepancies between V_{π_k} and Q_{π_k}. We know from

Equations (4) and (5) that $V_\pi(s) = \mathbb{E}[Q_\pi(s, A_0)]$ for any π.[2] Thus, if we find some state-action pair (s, a) such that $Q_{\pi_k}(s, a) > V_{\pi_k}(s)$, we can create an improved policy π_{k+1} by setting $\pi_{k+1}(a, s) = P\{A_0 = a | S_0 = s\} = 1$ (since this is gauranteed to improve the expectation over A_0 above). For even better results, we want to repeat this process for all states and choose actions that *maximize* Q_{π_k} (rather than just beat V_{π_k}). This leads to the following policy improvement rule:

$$\pi_{k+1}(a, s) = \begin{cases} 1, & a = \arg\max_{a'} Q_{\pi_k}(s, a') \\ 0, & \text{otherwise} \end{cases} \qquad \forall s \in \mathcal{S} \tag{9}$$

This creates a greedy, deterministic policy that is gauranteed to out-perform the previous π_k.

Keep in mind that by updating to $\pi_{k+1}(a, s)$, we've not only updated A_0, but also all future A_t's that occur at state s. This means that we have to recompute $V_{\pi_{k+1}}$, since it is an expectation over all future S_t's and A_t's. We do this by resorting to iterative policy evaluation again. After we've found $V_{\pi_{k+1}}$, the process repeats until we find π^*.

It turns out that there's some leniency in how we update our policy and value function estimates. For example, we can perform just one step of policy evaluation before improving the policy, rather than waiting for policy evaluation to converge over many steps. This variant is called *value iteration* and still converges to π^*. It can conveniently be written as a one-line update rule:

$$V_{k+1}(s) = \max_{a'} Q_k(s, a') \tag{10}$$

Another option is to only update the value function and policy for a subset of states per iteration, rather than all of $\mathcal{S}$. This is called *asynchronous* dynamic programming and it also converges to π^*, as long as every state is updated infinitely many times in the limit of total iterations. These algorithms rely on a more general notion of alternating between policy evaluation and policy improvement, which is termed *generalized policy iteration*.

2.3 Monte Carlo and Temporal-Difference

Dynamic programming finds an optimal policy, but it has a major drawback. Recall that when performing a Bellman backup (Equation (8)), we needed to compute the expectation over S_1 and A_0. This requires knowing the MDP's transition probabilities, $\mathcal{T}$, which are often unknown in the real world. In this section, we'll look at how to extend dynamic programming for finding π^* without knowning $\mathcal{T}$, by using sampled trajectories and rewards.

Before proceeding, we'll want to clarify what we mean by sampling. So far, we have defined sampling using the random variables S_0, A_0, etc. which are derived from the MDP described earlier. Since the MDP is a model for some actual agent-environment interaction, we can use the actual system as a source

[2] This relationship also gives us a way to compute V directly from Q in general

of samples. More specifically, we can initialize the actual agent at some start state with some policy and run the system until termination to collect a sample trajectory. We can then reset the agent to some potentially different state with a potentially different policy and repeat the process.

Now, our first step will be to find a sample-based (Monte Carlo) method that is analogous to policy evaluation. We can start by defining a sample *return* based on the Bellman equation:

$$\tilde{V}_{\pi,s_t} = R(s_t, a_t, s_{t+1}) + \gamma \tilde{V}_{\pi,s_{t+1}} \tag{11}$$

$$\tilde{Q}_{\pi,s_t,a_t} = R(s_t, a_t, s_{t+1}) + \gamma \tilde{Q}_{\pi,s_{t+1},a_{t+1}} \tag{12}$$

where (s_t, a_t, s_{t+1}) is a subset of some sample trajectory τ. If we limit ourselves to finite trajectories, we can set:

$$\tilde{V}_{\pi,s_T} = R(s_{T-1}, a_{T-1}, s_T)$$
$$\tilde{Q}_{\pi,s_{T-1},a_{T-1}} = R(s_{T-1}, a_{T-1}, s_T)$$

and compute the remaining sample returns recursively. Notice that these returns are not functions of s or a: $s_t = s_{t'} \implies \tilde{V}_{\pi,s_t} = \tilde{V}_{\pi,s_{t'}}$. From here, we can sample multiple trajectories and bin the resulting $\tilde{V}_\pi$'s and $\tilde{Q}_\pi$'s based on state and action:

$$\tilde{V}_{\pi,s} = \{\tilde{V}_{\pi,s'} : s' = s\}$$
$$\tilde{Q}_{\pi,s,a} = \{\tilde{Q}_{\pi,s',a'} : s' = s, a' = a\}$$

Finally, we can estimate V_π and Q_π by averaging over these bins:

$$V_\pi(s) \approx \frac{1}{|\tilde{V}_{\pi,s}|} \sum_{\tilde{V} \in \tilde{V}_{\pi,s}} \tilde{V} \qquad \forall s \in \mathcal{S} \tag{13}$$

$$Q_\pi(s,a) \approx \frac{1}{|\tilde{Q}_{\pi,s,a}|} \sum_{\tilde{Q} \in \tilde{Q}_{\pi,s,a}} \tilde{Q} \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{14}$$

Without explicitly writing out $\mathcal{T}$ (or π for that matter), this method gives us an estimate of V_π and Q_π. As long as we sample returns from every state-action pair infinitely many times in the limit, this estimate converges to the functions' true values.

Next, we can incorporate these value function estimates into policy iteration. We still want to update the policy based on Q_π and V_π, but we also want to make sure that the resulting policy can still sample every state-action pair so that our estimates converge. We can assume that $\mathcal{T}$ has at least one non-zero transition probability into every state (otherwise, why would it be in the state-space?), so we know that we can potentially get samples from every state. However, we can't assume that our policy gets samples from every state-action pair. In fact, if we always choose actions deterministically, then most state-actions pairs will never be visited more than a few times.

To counteract this, we can modify our policy improvement step slightly. Instead of updating to a completely deterministic policy, we will update to an ϵ-*greedy* policy:

$$\pi_{k+1}(a,s) = \begin{cases} 1-\epsilon, & a = \arg\max_{a'} Q_{\pi_k}(s,a') \\ \frac{\epsilon}{|\mathcal{A}|-1}, & \text{otherwise} \end{cases} \qquad \forall s \in \mathcal{S}$$

This way, our sampling method is gauranteed to visit every state-action pair infinitely many times in the limit. With this in place, we can implement a form of generalized policy improvement by alternating between sample-based policy evaluation and ϵ-greedy policy improvement. This converges to the slightly suboptimal ϵ-greedy approximation of π^*.

Even though ϵ-greedy policies visit every state-action pair, they don't necessarily visit them efficiently. The Q_π estimates for low-probability state-action pairs might stay inaccurate even after sampling many trajectories. In general, choosing π' so that it simultaneously improves on π and generates informative sample returns is a non-trivial problem. This is called the *Exploration-Exploitation* dilemma and has been studied extensively in the context of bandit problems. That literature suggests several alternatives to ϵ-greedy policy improvement, but we won't cover those extensively here.

Now, we will look at a valuable extension of the Monte Carlo method from earlier, *off-policy learning*. So far, we have collected returns by sampling trajectories from policy π and averaging them to calculate V_π and Q_π. However, these sample returns can also be combined with *importance sampling* to calculate V and Q for other policies. The basic idea is that we can represent the expectation of one random variable as the expectation of another:

$$\mathbb{E}[X] = \int_{\mathbb{R}} u f_X(u) du = \int_{\mathbb{R}} u f_X(u) \frac{f_Y(u)}{f_Y(u)} du = \int_{\mathbb{R}} u \frac{f_X(u)}{f_Y(u)} f_Y(u) du \tag{15}$$

This is accomplished by rescaling each u by $\frac{f_X(u)}{f_Y(u)}$, the *likelihood-ratio* between X and Y. The same idea applies to Monte Carlo estimation: if we want to estimate V_μ and Q_μ for some arbitrary *target* policy μ, but only possess sample returns from the *behavior* policy π, we can rescale each sample return based on its likelihood-ratio under the two policies:

$$\tilde{V}_{\mu,s_t} = \frac{p_\mu(\tilde{V}_{\pi,s_t})}{p_\pi(\tilde{V}_{\pi,s_t})} \tilde{V}_{\pi,s_t} \qquad \tilde{Q}_{\mu,s_t,a_t} = \frac{p_\mu(\tilde{Q}_{\pi,s_t,a_t})}{p_\pi(\tilde{Q}_{\pi,s_t,a_t})} \tilde{Q}_{\pi,s_t,a_t}$$

and then average over all returns as in Equations (13) and (14). Calculating p_μ and p_π for a sample return is fairly similar to calculating $p(\tau)$ in Equation (2). The main difference is that we only need to consider the trajectory starting from s_t, since past states and actions do not influence the current return. We also need to replace any instance of π with μ when calculating p_μ.

$$\frac{p_\mu(\tilde{V}_{\pi,s_t})}{p_\pi(\tilde{V}_{\pi,s_t})} = \frac{\prod_{t'=t}^{T-1} \mathcal{T}(s_{t'+1}, a_{t'}, s_{t'}) \mu(a_{t'}, s_{t'})}{\prod_{t'=t}^{T-1} \mathcal{T}(s_{t'+1}, a_{t'}, s_{t'}) \pi(a_{t'}, s_{t'})} = \frac{\prod_{t'=t}^{T-1} \mu(a_{t'}, s_{t'})}{\prod_{t'=t}^{T-1} \pi(a_{t'}, s_{t'})} \tag{16}$$

Luckily, the transition probabilities (which we don't know) cancel out in this expression. The likelihood-ratio is nearly identical for $\tilde{Q}_{\pi,s_t,a_t}$, except that the products start from $t' = t + 1$.

By using importance sampling to learn a target policy, we can decouple our sampling method from our policy iteration method, and generally get more use out of our sample returns. Besides the simple variant of importance sampling presented above, there are a number of other off-policy learning methods that acheive a similar effect, some of which will be discussed later.

To conclude this section, we will shore up two major weaknesses of the Monte Carlo method, and in the process present some of the earliest reinforcement learning algorithms developed. When we defined sample returns earlier, we needed to limit ourselves to finite trajectories. Otherwise, Equations (11) and (12) would recur indefinitely. We also needed to sum over all returns at once when estimating V_π and Q_π in Equations (13) and (14). Over time, we would need more and more memory to improve these estimates!

To solve these issues, we would need a method that *incrementally* updates our estimates using only *local* trajectory information. The way to do this is with *temporal-difference* (TD) learning. To establish this approach, first we'll define a general method for incrementally updating a mean $M_k = \frac{1}{k}\sum_{i=1}^{k} w_i$ with the newest sample, w_k:

$$\begin{aligned} M_k &= \frac{1}{k}\sum_{i=1}^{k} w_i \\ &= \frac{1}{k}\left(w_k + \sum_{i=1}^{k-1} w_i\right) \\ &= \frac{1}{k}\left(w_k + (k-1)M_{k-1}\right) \\ &= \frac{1}{k}\left(w_k + kM_{k-1} - M_{k-1}\right) \\ &= M_{k-1} + \frac{1}{k}\left(w_k - M_{k-1}\right) \end{aligned}$$

To make the estimate from Equation (17) robust in case the sampling process is non-stationary, we can change $\frac{1}{n}$ to an arbitrary constant α, weighting new samples more than older samples.

$$M_k = M_{k-1} + \alpha\left(w_k - M_{k-1}\right) \tag{17}$$

This puts more weight on recent samples in the average. At this point, we could replace w_i with our $\tilde{V}$'s and $\tilde{Q}$'s but we would still need to compute each of those from complete trajectories. Instead, we'll replace w_i with TD returns:

$$\hat{V}_{k,s_t} = R(s_t, a_t, s_{t+1}) + \gamma\hat{V}_k(s_{t+1}) \tag{18}$$

$$\hat{Q}_{k,s_t,a_t} = R(s_t, a_t, s_{t+1}) + \gamma\hat{Q}_k(s_{t+1}, a_{t+1}) \tag{19}$$

Instead of computing a recursive relationship between returns like in Equations (11) and (12), this approach plugs the current and next sample state-action pair into our *current estimate* of the value functions, $\hat{V}_k(s)$ and $\hat{Q}_k(s,a)$. We can then update these estimates using Equation (17):

$$\hat{V}_{k+1}(s) = \hat{V}_k(s) + \alpha\left(\hat{V}_{k,s_t} - \hat{V}_k(s)\right)$$
$$\hat{Q}_{k+1}(s,a) = \hat{Q}_k(s,a) + \alpha\left(\hat{Q}_{k,s_t,a_t} - \hat{Q}_k(s,a)\right)$$

Since each iteration of $\hat{V}_k(s)$ and $\hat{Q}_k(s)$ corresponds to one new state s_t, we can combine k and t into one index. We can also substitute-in the TD return expression, giving us the following one-line update rules:

$$\hat{V}_{t+1}(s) = \hat{V}_t(s) + \alpha\left(R(s_t,a_t,s_{t+1}) + \gamma\hat{V}_t(s_{t+1}) - \hat{V}_t(s)\right) \tag{20}$$
$$\hat{Q}_{t+1}(s,a) = \hat{Q}_t(s,a) + \alpha\left(R(s_t,a_t,s_{t+1}) + \gamma\hat{Q}_t(s_{t+1},a_{t+1}) - \hat{Q}_t(s,a)\right) \tag{21}$$

This TD update effectively combines the sampling techniques described in this section with the Bellman backup from Equation (8). Miraculously, this update rule not only solves the two weaknesses described above, but also converges to an optimal policy faster than the Monte Carlo Method (the proof is currently omitted here).

The TD update forms the backbone of two classic reinforcement learning algorithms: Sarsa and Q-learning. Sarsa is an on-policy algorithm that estimates $Q(s,a)$ precisely using Equation (21) as its update rule. After each update to $\hat{Q}(s,a)$, Sarsa generates the corresponding ϵ-greedy policy, samples one trajectory step, and then repeats. Q-learning, on the other hand, is an off-policy algorithm that uses a very similar update rule:

$$\hat{Q}_{t+1}(s,a) = \hat{Q}_t(s,a) + \alpha\left(R(s_t,a_t,s_{t+1}) + \gamma\max_{a'}\hat{Q}_t(s_{t+1},a') - \hat{Q}_t(s,a)\right) \tag{22}$$

The only difference is the maximum in the TD return. This means that the return is calculated as though we took an action following a greedy deterministic target policy, rather than using the sampled action a_{t+1}. After updating $\hat{Q}(s,a)$, the algorithm updates its policy, samples a trajectory step, and repeats. Since the TD update learns the greedy deterministic policy, we have more leeway in how we update our behavior policy for sampling (but often times we still use an ϵ-greedy policy like in Sarsa).

Besides the basic form of TD returns presented here there are a number of extensions that we will discuss briefly. The main idea is that we can plug a longer sequence of sample state-action pairs into an *unrolled* version of the value function equations, for example:

$$\hat{V}_{k,s_t} = R(s_t,a_t,s_{t+1}) + \gamma R(s_{t+1},a_{t+1},s_{t+2}) + \gamma^2\hat{V}_k(s_{t+2})$$
$$\hat{Q}_{k,s_t,a_t} = R(s_t,a_t,s_{t+1}) + \gamma R(s_{t+1},a_{t+1},s_{t+2}) + \gamma^2\hat{Q}_k(s_{t+2},a_{t+2})$$

We can unroll our returns to any desired length and then implement the rest of the reinforcement learning algorithm as described earlier. Note that for offline algorithms, this return would also need to be weighted using importance sampling, unlike in the simple case (details omitted). These unrolled TD returns can potentially improve the convergence of the reinforcement learning algorithm.

We can take this idea further by averaging returns of different lengths. More specifically, if we have some (possibly endless) sample trajectory $(s_0, a_0, \ldots)$, we could compute the simple TD return using (s_0, a_0, s_1, a_1), then compute an unrolled TD return using $(s_0, a_0, s_1, a_1, s_2, a_2)$, then compute a twice-unrolled TD return using $(s_0, a_0, s_1, a_1, s_2, a_2, s_3, a_3)$, and so forth. As we gather more samples, we could average over more unrolled TD returns and hopefully improve our value function estimates faster than using only simple TD returns.

λ-returns are a particular averaging scheme for this sequence of unrolled TD returns. If we denote an n-step unrolled TD return as $\hat{V}_{k,s_t,n}$ (the $\hat{Q}$ case is identical and omitted), then a λ-return is defined as:

$$\hat{V}^{\lambda}_{k,s_t} := (1-\lambda)\sum_{n=0}^{\infty} \lambda^n \hat{V}_{k,s_t,n} \tag{23}$$

This is a weighted average over all possible unrolled TD returns starting in state s_t, with weights decaying based on the factor $\lambda \in [0, 1]$. TD learning using λ-returns is called TD(λ), with TD(0) being equivalent to Equations (18) and (19) and TD(1) being equivalent to the Monte Carlo method from earlier.

While at first glance this approach may seem like it would require infinite memory, it turns out that there is an elegant approach to approximating λ-returns in an online manner using *eligibility traces*. Eligibility traces work by re-using simple TD returns rather than explicitly unrolling the returns in Equation (23). At each time step, we store the TD *error*, $\hat{V}_{k,s_t} - \hat{V}_k(s)$ for state $s = s_t$. The TD error is used to update $V_k(s)$ not just once, but at every time step until state s is revisited. The TD error is scaled by the eligibility trace, which decays by $\gamma\lambda$ at each time step. Once state s is revisited, the TD error is updated and the trace is incremented by 1 (in the simplest case). This gives us a simple method for incorporating information across different durations into reinforcement learning algorithms.

2.4 Summary

In this section, we defined a problem that involves optimizing an agent's interaction with its environment. We defined this interaction using an MDP, which consists of a set of states, actions, transitions probabilities, and transition rewards with discounting. We then presented dynamic programming as an iterative method for learning the agent's optimal policy given information about the MDP. The key insight was that the MDP's value functions could be used to choose improved policies, and that we could progressively estimate these value functions using the Bellman equation as an update rule. Afterwards, we presented a Monte Carlo method that also finds an optimal policy but doesn't

require knowledge of the MDP's transition probabilities. We computed returns by using sampled states and actions as inputs to the value functions, and averaged over them to approximate the true value functions. We also introduced the idea of exploration and why we cannot use deterministic policies for sampling. We also introduced off-policy learning as a way to decouple sampling from policy optimization. Finally, we combined the ideas from the dynamic programming and Monte Carlo methods into temporal-difference learning, which computes returns by inputting samples into the Bellman equation and iteratively updating an average over time. We presented two classic reinforcement learning algorithms that use temporal-difference returns, and also presented elgibility traces as an extension to temporal-difference learning that approximates using multi-duration returns.

2.5 Motivations for Modern Work

Having described the fundamental reinforcement learning problem, we have most of the information we need to understand many of the current research directions available in reinforcement learning. It turns out that there are quite a few limitations to the framework and algorithms described so far. Some of these limitations are:

Scalability: The methods described so far are *tabular*. We were able to enumerate every possible state-action pair, and update the policy with respect to each one individually. As we scale the number of states and actions in our problem this becomes less practical. If states and actions are continuous, we need a different approach.

Time: The MDP used discrete, uniform time steps. This might not be the best way to represent problems involving continuous dynamics or behavior that spans multiple time scales.

Dynamics: The sample-based approaches didn't make any use of transition probabilities. It may be possible to learn or use a pre-existing estimate of the MDP's dynamics within our reinforcement learning framework, but so far it's not clear how we could incorporate this.

Non-stationary MDPs: TD updates only address non-stationary MDPs by weighting recent samples more heavily using a constant coefficient. There are many problems where this is insufficient, and we need a more informed approach to non-stationary dynamics.

Unknown State: The agent might not have access to state information for the MDP. It might have noisy or incomplete observations to work with instead.

Exploration: We have already mentioned the exploration-exploitation dilemma earlier, but there are a handful of other problems associated with exploration, for example identifying and dealing with absorbing states that cannot be escaped with the actions available to the agent.

Rewards: We have assumed that we can attribute a reward to every MDP transition using a reward function. It's not clear whether it is feasible to do this for complex problems or whether the reward function corresponds with the behavior we desire from the agent.

Many research efforts try to address one or more of these limitations, in one way or another. We will look at some specific areas in the following section.

References

[1] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *CoRR*, abs/1806.10293, 2018.

[2] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *J. Mach. Learn. Res.*, 17(1):1334–1373, Jan. 2016.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4] OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –, Jan 2016. Article.

[6] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[7] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.